Document Number	ENG-1234
Revision	0.1
Author	Hari Lalgudi
Project Manager	Steve Kufer

Salsa4 ACL

Software Unit Design Specification

Project Headline

This document describes the software design for the input ACL processing with Salsa4 ASIC used in the TTM48 engine based linecards e.g. Gigabit Ethernet, Fast Ethernet.

Reviewers

Department	Name
Development Engineering	Steve Kufer, Manager, Software Development
Development Test Engineering	
Impacted Groups	parser-police

Modification History

Rev	Date	Originator	Comment
0.1	07/15/99	Hari Lalgudi	Initial Release

Definitions

This section defines words, acronyms, and actions which may not be readily understood.

SALSA4	ASIC on the GSR TTM48 Linecards that support ACL lookup in hardware Refer to ENG-35918 for complete description. In this document Salsa by default will refer to Salsa4 only.
ACL	Access control lists used for packet classification and filtering. Each ACL has number of items, each specifying permit/deny action for packets matching the source, destination
Ex-ACL	Extended ACL, which matches more fields in IP header than standard ACL (source IP address). Fields include Destination IP address, source and destination TCP/UDP ports, protocol type.
TTM48	Time to Market OC48 engine and line cards built on this Engine(also refered as TTM)

GE	Gigabit Ethernet GSR linecard with TTM48 engine and Salsa4 ASIC (unless otherwise specified)
FE	8 port Fast Ethernet GSR linecard with TTM48 engine and Salsa4 ASIC (unless otherwise specified)
S-ACL	ACL lookup done by Salsa4
FIB, CEF	Forwarding Information Base or Cisco Express Forwarding, which is an mtrie used for lookup(based on destination IP address) to forward packets
CAR	Committed Access Rate. It is an IOS feature that may use ACLs to match packets and verify conform/exceed rate limits and perform actions such as permitting, dropping or marking the packet.

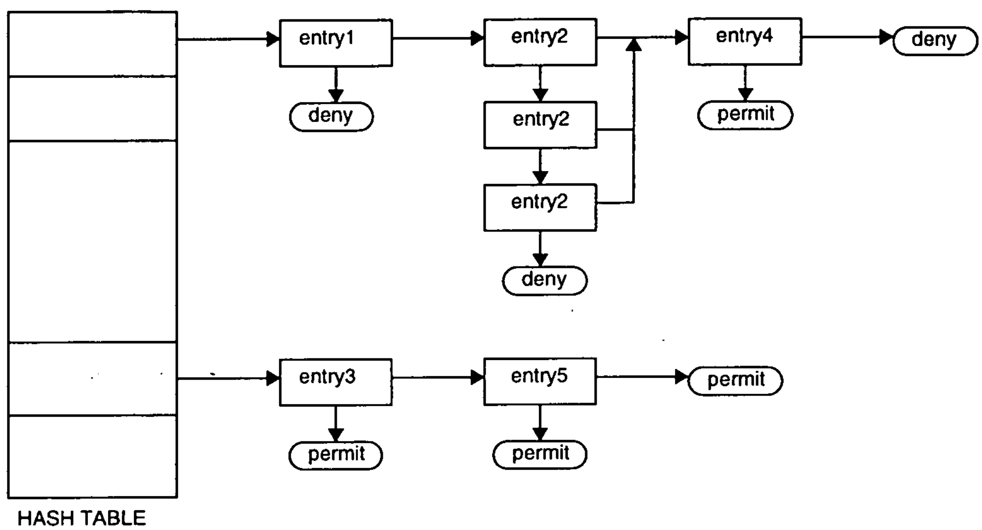
1.0 Problem Definition

TTM48 linecards without Salsa4 ACL hardware support, do extended ACL lookup in software. This CPU intensive operation results in a performance degradation from linerate (650 Kpps) without ACLs to 32Kpps with 512 ACL items. With Turbo or compiled ACLs in software, performance drop is from 650Kpps to a constant 270 Kpps (independant of the number of ACL items).

TTM ACL hardware solution needs to (a) presetve the order of items and checks in ACLs (b) provide full filtering or classification functionality(e.g. source and destination IP address, TOS, TCP/UDP port ranges) currently supported in software (c) higher performance than Turbo ACLs (270Kpps) for 10-1000 ACL items..

Salsa4(ENG-31157) solution is a hardware engine that traverses a hashed linked list of nodes(figure below), that is equivalent to the extended ACL, to permit or deny the packet. Each node(64 bits) is an instruction to Salsa4 to compare one or more fields(source/destination IP address, TCP/UDP port) in the incoming packet against an ACL parameter, and find the next match or miss nodes. The last(Stop) node specifies permit/deny value and more information for software to execute any other actions(e.g. ACL accounting, ACL logging or TCP flags checks).

f



Salsa4 ACL support is only for input ACLs, since Salsa4 does not do ACL lookup after FIB lookup is done when output adjacency/port areidentified. Also, Salsa4 does not support subinterface ACL, since subinterface identification (e.g. based on VLAN) is difficult in the given hardware.

2.0 Design Considerations

S-ACL(Salsa4 ACL) software tasks are:

1. Parse IOS ACL configuration to create the hash table and list of nodes.

2. Switching path code to process S-ACL node information(e.g. permit/deny, ACL logging, check TCP flags)

3. Handle error conditions(e.g. maximum number of nodes read) and continue software lookup on the ACL hash list in selected cases.

4. Provide user configurations to enable S-ACLs and to do performance tuning by weighting selection of hash bits.

2.1 Requirements/Constraints on parsing IOS ACL to Hash Table and Nodes

S-ACL nodes should preserve order of ACL item checks. S-ACL performance crucially depends on the average number of nodes Salsa4 processes before coming to the Stop node. This depends on

(a) the bits selected for hashing

(b) the parsing algorithm for creating nodes from ACLs.

(c) traffic pattern to different Stop nodes.

2.2 Constraints on Switching Code for S-ACL support

1. There should be no performance impact on GSR linecards that do not use Salsa4 ASIC.

2. TTM48 linecards can achieve 650Kpps by offloading FIB lookup in Salsa and the software saving cycles by not caching and invalidating packet headers in the L2 cache. To achieve performance better than 270Kpps(Turbo-ACLs), software should only cache/invalidate packet headers in L2 cache only (a) if Salsa4 does not find a valid(permit/deny) Stop node or (b) need to apply other features(input CAR/accounting).

2.3 S-ACL vs Turbo-ACL

Although Turbo-ACL gives 270Kpps on GSR LC with input ACLs irrespective of the number of ACLs, , the performance drop from 650 Kpps for TTM48 LCs is significant. S-ACL performance is dependant on the number of ACL items (which determines number of ACL nodes). We do not provide automatic selection of Turbo-ACL or S-ACLs. Here are the guidelines for the user for manual selection via configuration:.

(a) For output ACLs, use Turbo-ACLs

(b) For input ACLs where number of input items leads to performance < 270Kpps, use Turbo-ACLs, otherwise use S-ACLs.

2.4 Separation of hardware dependant and hardware independant modules

S-ACL hash tables and trees should be made hardware independant so that pure software lookups without hardware assist is possible This is useful for (a) linecards without Salsa4 and (b) ACL lookup for packet classification purposes other than filtering (e.g. ACL based CAR)

2.5 Memory requirements for S-ACL

The Match/Miss address in each ACL node is derived from the "node base"(10b) and match/miss "offset"(8b). The algorithm to create ACL nodes needs to make sure that the "node base" is the same for the match and miss nodes. We guarantee this by allocating 2^18 nodes for each S-ACL node memory.

Thus the memory requirements for S-ACL on 8 port FE LC is (8 * 2^18 * 8B) = 8MB

3.0 Functional Structure

The S-ACL software is separated into Salsa4 specific module and hardware independant ACL Hash Table module.

The Salsa4 module sets up Salsa4 registers to enable/disable ACL lookup, setup hash table addresses, select hash tables based on multi ported linecards (e.g. FE), and to keep history of errors during S-ACL lookup.

The ACL Hash Table module however needs parameters from the hardware dependant module to allocate memory that is consistent with hardware requirements (e.g. 8B aligned or hash table next to node memory map)

The current scheme of sending ACL config is sent from RP to LC via FIB IPC messages is unchanged. While parsing interface ACL config on LC, if S-ACL interface is affected, then we do the following:

(1) disable Salsa4 from using S-ACL hash tables.

(2) clear the current S-ACL hash table and node memory map.

(3) recalculate hash key and insert hash nodes for each ACL item. For multiported linecards(FE) since there is only one hash key for all hash tables, this implies recalculating hash key based on all FE interface S-ACL configurations and inserting ACL nodes for all FE interface ACL items.

4.0 Packet Flow

We need to add support for S-ACL in the software switching code without affecting performance of existing linecards that do not use Salsa4. Switching vectors introduced by Netflow on GSR(12.0(6)S) allows us to add new features in the switching path without incurring the runtime performance. Currently, there are 4 switching vectors for TTM path: no flow/no feature, flow/no feature, no flow/feature, flow and feature (feature refers to FIB/CEF features such as input/output ACLs or CAR or accounting).

S-ACL support makes this 8 switching vectors: no S-ACL/no flow/no feature, no S-ACL/flow/no feature, no S-ACL/no flow/feature, no S-ACL/flow/feature, S-ACL/no flow/no feature, S-ACL/flow/no feature, S-ACL/no flow/feature, S-ACL/flow/feature. We select S-ACL switching vectors if we find Salsa-4 version (Salsa L3 Asic Id Register >= 4).

In S-ACL switching vectors, if Salsa4 gets a valid FIB leaf, we read the S-ACL registers (while still working in L2 uncached mode for accessing bhdr and IP ptrs). If there are no S-ACL errors (S-ACL status register), and no other feature checks (e.g. input/output CAR/accounting) need to be done, we switch the packet in uncached mode. Else, we cache the packet, check other features, switch the packet and invalidate the L2 cache (and incurring the performance penalty of dropping to 300-400Kpps).

If S-ACL is successful, we need to disable input ACL checks in the FIB feature path. This is done by setting the platform specific FIB control block parameter to skip input ACL checks.

5.0 Salsa4 error handling

Salsa4 may report the following types of errors in S-ACL processing:

- ACL Engine Maxed out error: Salsa4 did not reach a STOP node(permit/deny) at the end of maximum number of lookups (software configurable register). The DRAM Catastrophic Error Address register gives the address information.
- ACL Engine uncorrected ECC error: Salsa4 encountered a uncorrectable ECC error during a DRAM read. The DRAM Catastrophic Error Address register gives the address information.
- ACL Engine out-of -range error:

S-ACL hash table errors(due to software bugs or insufficient memory) are reported by errmsgs. The following debug commands on the linecard allows further debugging:

- debug ip access hash
- debug ip access detail-hash

The following commands show the S-ACL hash table and nodes information. If the above debugs are turned off, only the "useful" nodes are display, otherwise all nodes in the hash table, including stop nodes are printed.

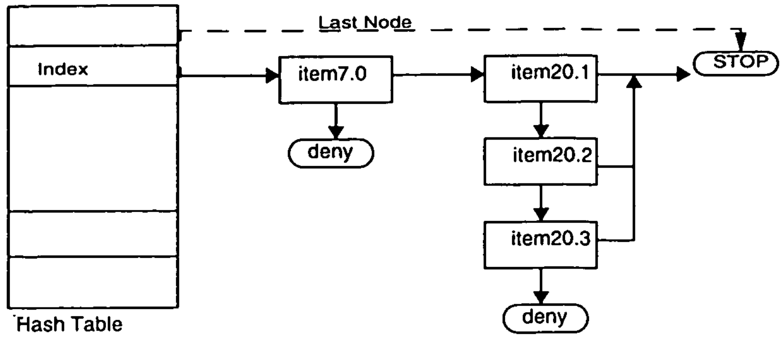
- show access hash <port-number> {nodes}

6.0 Algorithmic Description

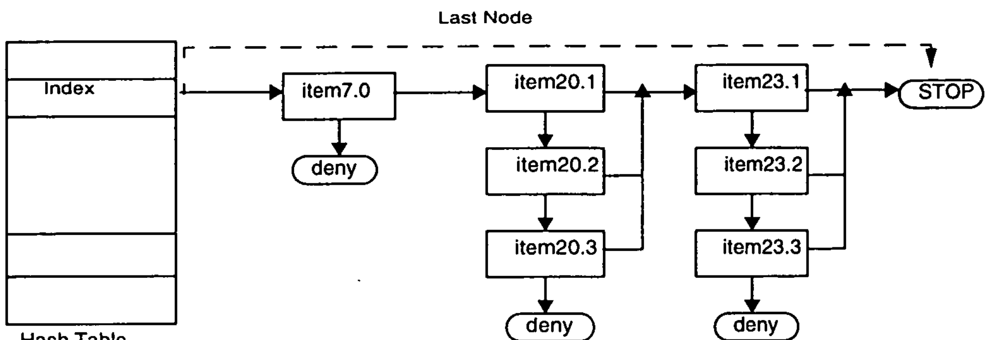
6.1 ACL Items mapping to ACL Nodes

Each ACL item may contain checks(exact match or range) for source IP address, destination IP address, protocol type, source and destination TCP/UDP port numbers and TOS. This translates to multiple ACL nodes, each node checking one value of the field (or multiple fields for TOS/Protocol). When we parse each item we set the "Miss" address in each of these nodes to a new Stop node, which is that index's "last node". Since we need to keep the ordering among ACL items, we use this "last node" as the starting node for the nextACL item that hashes into the same hash index

For example, item 7and 20 of an ACL, which map to the same hash index, are inserted in the table with a last stop node. Now we want to insert item 23, also hashing to the same index, starting from the last node.



Hash Table before Item 23 is inserted



Hash Table after Item 23 is inserted

6.2 Dynamic ACL Hash Key selection algorithm

Using the above ACL item to node allocation method, the worst performance is obtained when Salsa4 is made to walk all the ACL nodes for given hash index and terminates at the last node

We can reduce the distance to the last node by picking the right bits for ACL Hash Key selection.

Consider each ACL item to be represented as a "select" bitmask of <src_ip, dst_ip, protocol, src_port, dst_port, tos>, where a field's bit = 1 if the ACL item checks the field, else field's bit = 0.

If a field (e.g. bit 9 of destination IP address) is selected for determining the hash key, and if the corresponding bit in the item's "select" bitmask is 1, then the item's ACL nodes need to be inserted only in the hash index for which the bit is either 0 or 1 (depending on ACL item's value). If the "select" bitmask is 0, then the item's ACL nodes need to be inserted in all hash indexes where this bit is either 0 or 1.

Our algorithm maintains two counts, zero_count(for bit=0) and one_count(for bit=1), for each bit in the "select" bitmask. For each ACL item, we get a "per-item-wt" by multiplying the number of nodes for that ACL item with a per-ACL weight(default 1). If the "select" bitmask for that bit = 0, we add the "per-item-wt" to both zero_count and one_count. Else, if the item bit value in item is 0, we add the "per-item-wt" to zero_count only. Else if the item bit value is 1, we add "per-item-wt" to only one_count.

Worst case performance penalty of not selecting a bit in the hash key is given by max(zero_count, one_count) for that bit. Worst case performance of selecting the bit in the hash key is given by the min(zero_count, one_count) for that bit.

We seek to minimize the maximum performance penalty of selecting the bits. Hence we sort the min(zero_count, one_count) of all the bits and pick up the first "n" (for Salsa4, n = 10) bits. If there are two bits with the same min() value, the max() is used to differentiate the bits.

The per-ACL weights represents the amount of traffic hitting each ACL. These weights may be assigned by the user or calculated by the amount of traffic hitting the ACL as a percentage of all the traffic.

Traffic distribution may be skewed towards one protocol (e.g. HTTP with TCP/IP) and the ACLs may not represent this factor. We use a "per-field" weight to compare the performance penalty of selecting a field that may not be varying as much as other field.

7.0 SW Restrictions and Configuration

(1) S-ACL does not currently support ACL lookup for CAR or output ACLs.

(2) Currently support is only in 12.0S (no support is planned for 11.2GS releases)

(3) S-ACL is not supported for subinterface ACLs.

(4) S-ACL tree updates are not done on the fly, since Salsa4 may walk down old/unused paths.

8.0 HW Restrictions and Configuration

(1) For multiported TTM LCs (e.g. FE), we do not have multiple hash keys for the different hash tables. Since same hash key is used for multiple ports, a long ACL on one port may influence selection of hash key bits, thus influencing performance on other FE port using same hash key.

(2) Port information for selecting hash table is either in the INPUT_INFO field of BHDR or the 4B POS header(FE).

(3) Maximum lookup counter is implemented to prevent Salsa4 from getting lost in a corrupt ACL tree. This requires software to perform the rest of the lookups if a valid result requires more S-ACL checks.

(4) 8 port FE LC requires 4MB memory (8 chunks aligned at 256KB) for the 8 hardware ACL hash tables. If this memory is not available during initialization, S-ACL support may fail.

9.0 External Restrictions and Configuration

S-ACL support for GE is dependant on GSR product team approval for Salsa4 GE LC as an "enhanced" linecard or a new feature on existing linecards that may require a hardware upgrade of existing linecards.

10.0 Development Unit Testing

S-ACL unit test cases should verify the functionality for different types of ACL nodes(permit and deny) that can be generated by ACL configurations:

1. Source IP address with mask of 0.255.255.255, 0.0.255.255, 0.0.0.255 and 0.0.0.0
2. DestinationIP address with mask of 0.255.255.255, 0.0.255.255, 0.0.0.255 and 0.0.0.0
3. Protocol types: TCP, UDP, ICMP, IGMP, IGRP, EIGRP, IGRP, IPINIP, NOS, OSPF
4. TCP/UDP Source/Destination Port: port numbers with operations eq, lt, gt.
5. ICMP/IGMP Source/Destination port numbers with operations eq, lt, gt.
6. TOS values with combination of Source/Destination port numbers indicated above.

Other unit tests are to populate the ACL Hash table with at least one ACL item and check for validity. Example:

(a) Configure 1024 ACL items each with incrementing source/destination IP address, with alternating permit/deny values in each item.

(b) Send traffic for 1024 source/destination IP address and verify that Salsa drops/permits the packet and updates the right ACL item counters.

Other unit test is use "S-ACL debug" mode, where we configure 1000+ item list with incrementing source/ destination IP address or TCP ports or protocol fields. Configure "debug access switch" on the linecard that verifies the output of the S-ACL lookup with the conventional or Turbo-ACL lookup. Scripts then send traffic with incrementing source/destination IP addr or port or protocol numbers. Any mismatch between S-ACL and Turbo-ACL is reported via error msgs with packet dump and Salsa4 registers for further debugging.

References

Salsa4 ASIC Hardware Specification, ENG-31157, Faisal Haq

Appendixes

Specification Review

cisco Systems

Document Number	ENG-31157
Revision	0.2
Author	Faisal Haq
Project Manager	Kamal Avlani

Salsa4 ASIC

Hardware Specification

Project Headline

Summary of modifications in revision 4 of the Salsa switching ASIC

Approvals

Name	Approval Date
Faisal Haq HW Design Engineer	-
Kamal Avlani Program Manager	
Brad Wurtz HW Design Engineering Manager	-

Modification History

Rev	Date	Originator	Comment

Table of Contents

Section 1. Introduction

As the Salsa ASIC is being employed in multiple GSR 12000 Linecards, new requirements for this design have emerged that necessitate a spin. The following functionality is being considered for the spin, which is being called SALSA4:

- ECC Protection on Routing Table Memory (EDO) Interface
- Hardware based access list implementation.

Each of these features are discussed in much detail in the following sections. They have been implemented with the following assumptions:

- Pin-for-pin compatibility with previous revisions of Salsa. This limits Salsa4 to the original 503-pin EPBGA package and the LSI G10p process.
- Quick turnaround solution, that provides chips in hand within 3 months.
- Low risk changes that can be easily synthesized and verified within the allotted turnaround time.

Section 2 Functional Description of Modifications

2.1 Hardware-based access-list filtering

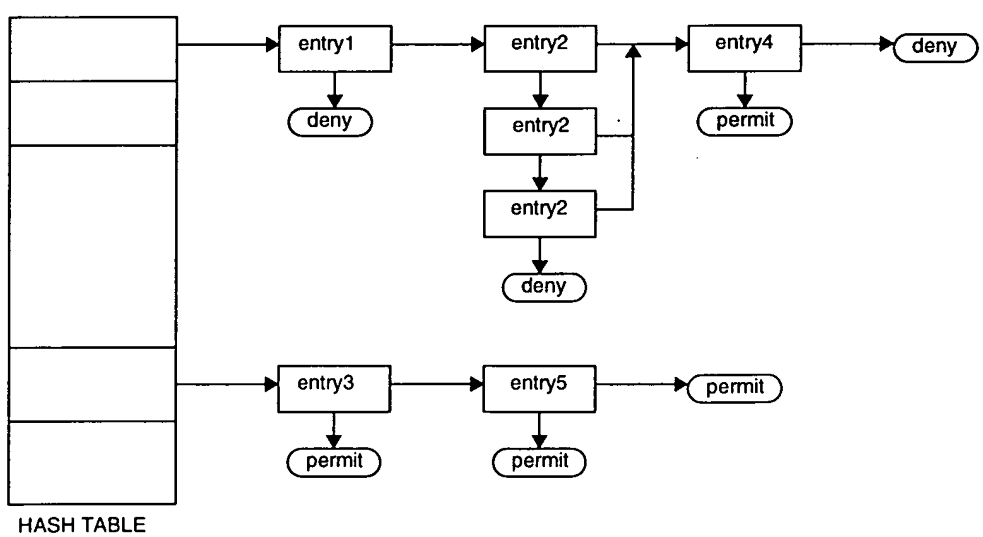
TTM based multi-port linecards (primarily DS3, Fast Ethernet and Gigabit etheret) are not expected to sustain an acceptable packet rate with software-only based extended access lists. The following performance has been demonstrated with "benchmark" access-lists obtained from ISPs.

Table 1: Performance of Extended Access Lists on exisitng GSR linecards, using a software-only approach

Access List	Linecard	L2 Cache?	Performance
Internal (256 entries)	Jaguar	Yes	38Kpps
Internal (512 entries)	Jaguar	Yes	32Kpps
Internal (256 entries)	QOC3 POS	No	35Kpps

2.1.1 The ACL Tree

Salsa4's solution is a hardware engine that traverses a linked-list implementation of the ACL (the ACL tree). There are several start points to the tree, and these are stored in a hash table that is accessed once per packet. There is one tree for each port of the linecard. Each entry in the user's access-list corresponds to one or more nodes in the ACLtree. The diagram below shows the implementaion:



Each node in the tree is an instruction to Salsa4, to compare one or more fields of the incoming packet, against a value determined by the ACL. The node is a 64bit entity of the following format:

32	2	4	10	8	8
operand	target	opcode	node base	match offset	miss offset

operand — This field is the value to be compared against, but its format depends on the type of comparison defined by the opcode bits. For a 32 bit compare, this field is the entire 32bit value to be compared against. For 16 bit compares, the upper 16bits is the value and the lower 16bits are mask bits.

target — The target field indicates to Salsa4, which field in the incoming packet is to be used for the compare. The decode is:

```
00    Source Address
01    Destination Address
10    The 32bit concatenation of {Dest.Port, Protocol,TOS}
11    The 32bit concatenation of {SourcePort, Protocol,TOS}
```

opcode — An opcode to define the type of compare:

```
0000  STOP. (See operand field for information on permit/deny)
0010  16bit compare of UPPER 2 BYTES of target
0011  16bit compare of LOWER 2 BYTES of target
0100  16bit greater-than compare of UPPER 2 BYTES of target
0101  16bit greater-than compare of LOWER 2 BYTES of target
0110  32bit compare against ACL Mask Register 0
0111  32bit compare against ACL Mask Register 1
1000  32bit compare against ACL Mask Register 2
1001  32bit compare against ACL Mask Register 3
1010  32bit compare against ACL Mask Register 4
1011  32bit compare against ACL Mask Register 5
1100  32bit compare against ACL Mask Register 6
1101  32bit compare against ACL Mask Register 7
1110  32bit compare against ACL Mask Register 8
1111  32bit compare against ACL Mask Register 9
```

node base — This is the 10bit base address for the next node addresses. This base is not to be confused with the PORT base address, discussed later.

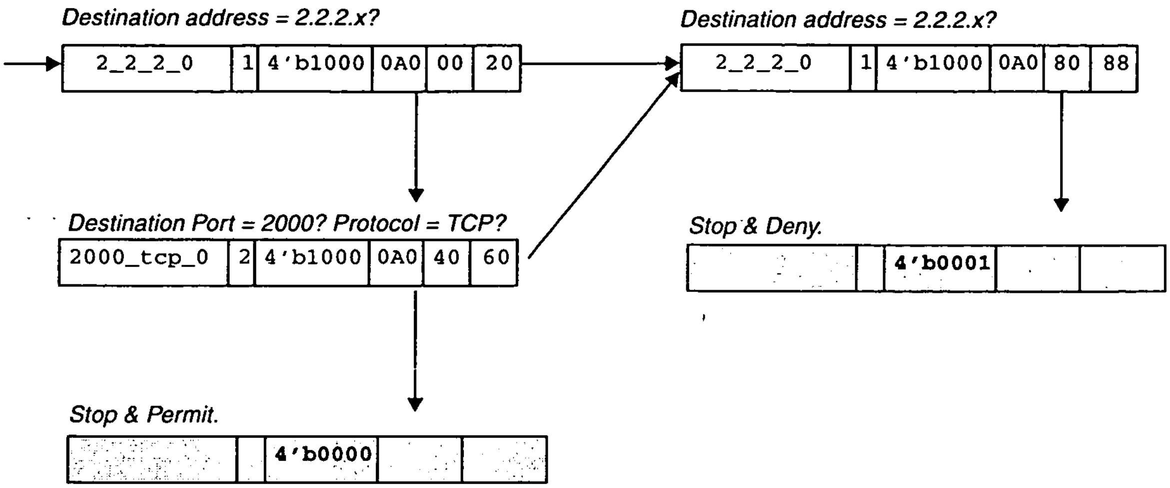
match address — Base offset for the next address, if the operand matches the target

miss address — Base offset for the next address, if the operand does NOT match the target

Thus, as an example the access-list entries..

```
access-list 101 permit tcp any 2.2.2.0  0.0.0.255  eq  2000
access-list 101 deny tcp any 2.2.2.0  0.0.0.255
```

would be implemented in the following manner, in the ACL tree:



Because of the inter-dependance of entries in the list, and as the example above also reveals, it is very important for software to maintain the user's order of ACL entries when constructing the tree. Without this, it is possible for packets to be permitted or denied incorrectly. It is suggested that software start from the bottom of the list and proceed upwards, inserting entries to the left of nodes in the tree,

2.1.2 A Perl program to determine Hash size

A perl script has been written to estimate the performance in pps, of access list lookups after they are hashed. The program converts actual ACL configurations into a hashed array of compare instructions, and provides relevant statistics on the size of trees stemming from the hash table buckets. The program has been written by Faisal Haq and Hari Lalgudi, and is included in an Appendex at the end of this document.

Based on a large database of customer access-lists (used for the design of the Toaster ASIC), and on ACL configurations sent to the Optical Internetworking BU from AOL, results from this program suggest a 1024-bucket hash table as being most optimal. Larger hash tables tend to reduce the overall depth of trees stemming from the buckets. Smaller hash tables consume less memory and are quicker to update. It was found that the incremental reduction in average tree depth was lowest in going from 1024 to 4096-bucket hash tables. Thus we havev chosen 1024 as the size of the hash table, implying a 10-bit hash key.

2.1.3 Determining which fields are used in Hash Key

Many of the entries in a list are mutually exclusive and it is this characteristic that allows for a performance improvement through the use of a hash table. For instance, we could group all entries in a list by the first ten bits of the destination address (still preserving order within groups!!). Then for each packet, Salsa4 would use the first ten bits of the destination address to lookup a 1024-deep hash table that would then point it to all related ACL entries.

Grouping by destination address as it turns out may not be optimal for two reasons. 1) Entries with an "any" in the destination address field would have to be replicated on all 1024 buckets, which would make the table large and impact performance for every packet being hashed. 2) Depending on where it is deployed, the linecard may receive packets destined for only a few addresses, and Salsa4 would not use the hash table uniformly over all buckets.

After many iterations with various keys, it was determined that no one key would prove optimal across the entrie suite of customer ACLs. It was then decided to go with a software-selcectable key that was customized for the ACL entered by the customer. The criteria for selection is for bits that occurr most frequently in the ACLs and occur evenly as a 1 and a 0.

So in the example below,

```
access-list 101 permit tcp any 2.2.2.1
access-list 101 deny   tcp any 2.2.2.254
```

the last 8 bits of the destination address would be among the bits chosen for the key since they occur in both entries, once with a value of 0 and an once with a value of 1. In contrast, the remaining bits of the destination address occur only as one value. This algorithm was writen into the perl program mentioned in Section 2.1.2.

The hardware design in Salsa4 allows software to select the fields used in the key, on a per-bit basis.

2.1.4 ACL Search Start and Stop Conditions

Once a packet is received in Salsa4, the MTRIE lookup logic will commence. It will complete within 4 DRAM lookups. Once the state-machine controlling this activity has returned to an IDLE state, the ACL lookup state machine shall begin. It generates a hash key using bits from the incoming packet, as specified by the software-programmed registers.

The hash key points Salsa to a chain of ACL nodes. An internal maximum-counter is clocked on each node lookup. Should that counter reach 0, the ACL lookup is terminated and the results are layed out in the "ACL Engine Status Register" on page 47. However, if a STOP node is found at or before then, the ACL lookup is ended and the results are presented in this register.

A STOP node has a special format, as shown below:

32	2	4	13	13
operand		0000 STOP		

Table 2: Decode of information stored in the OPERAND field of a STOP node.

Bits	Description
[31:4]	ACL number associated with this STOP
[3]	Deny/Permit 0 - Deny 1 - Permit

Table 2: Decode of information stored in the OPERAND field of a STOP node.

Bits	Description
[2:0]	Reserved, 0's

The contents of the operand field are copied into the "ACL Engine Status Register".

2.1.5 Limitations of the Salsa4 Approach

- The hardware to lookup the hash table and then follow the ACL tree will be added as an extension of the MTRIE lookup engine on the Rx BMA interface. This produces the most significant limitation of the Salsa4 approach -- it can ONLY work on the Rx side. Output port based access-lists would still have to be done in software.
- Salsa4 must handle per port access lists. Support for upto 8 individual ports is provided.
- Port information for each packet can be found in either the INPUT_INFO field of the BHDR or in the 4-byte POS header (as used by the Eiffel project). These are the only 2 places that Salsa will look for port information.
- A maximum lookups counter will be implemented as well, to prevent Salsa4 from getting lost in a corrupted ACL tree. This programmeable counter maxes at 1024 lookups, after which the packet is presented to the processor along with the address of the most recent access. Our tests show a number of cases where the lookups exceed 32. Beyond this point it is better to have software perform the lookups, since it may have better performance than hardware, given the L2 cache onn the board.
- The node format of a node-base, with 8bit match and miss offsets means that miss/match nodes stemming from the current node, MUST be within 2K Bytes of each other.
- The ACL tree size per port is limited to 2^{11}. With 18 bits (excludes 3 LSbits, since nodes are on 8byte address boundaries) remaining for addressing, we have a per-port limitation of 256K nodes. Note: that is 256K nodes, NOT acl-entries!!. Note that each ACL entrie may be implemented in as many as 4 nodes.
- ACL entries with protocols not using source/desintation ports (icmp, igmp, eigrp, gre, igrp, ipinip, nos and ospf.) will be treated as having "don't care" ports. This is because Salsa will not recognize these types of protocols and will treat them like tcp/udp. As a result these entries will be replicated over any buckets that are represented by PROTOCOL bits in the key. (That was pretty confusing)
- Tree updates on the fly if needed, would require special handling by software to make sure Salsa4 isn't sent down old or null branches.

2.1.6 Implementation in Hardware

The implementation is best summarized in the diagrams that follow. It involves an internal state machine that iteratively walks through the ACL tree, starting first at a hash table, and subsequently following either the match or miss address of each ACL node it encounters. A maximum of 1024 ACL nodes will be iteratively read, after which the state machine stops and delivers the last read node. Software will then continue the search as needed. Ofcourse, if the state machine encounters a "Stop with Deny" or "Stop with Permit" node, that too will terminate the search.

The area for this logic is approx twice that of the MTRIE lookup engine in Salsa4. That area was approx 8000 gates. There should not be any difficulty fitting this additional logic into the existing floorplan.

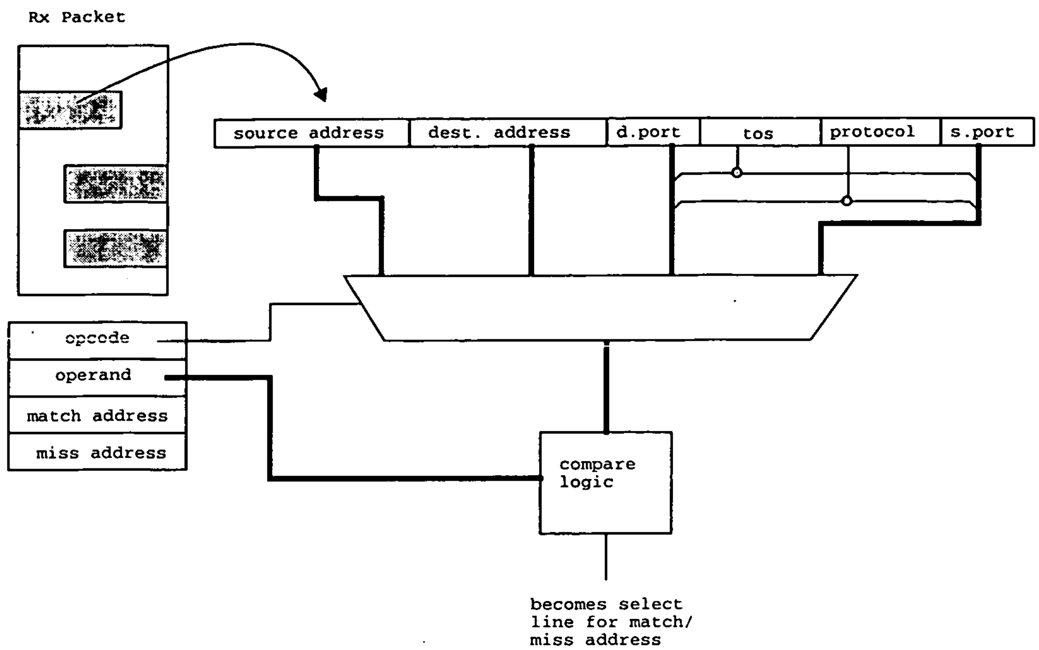


Figure 1: Comparison logic used to match/miss on a node's operand

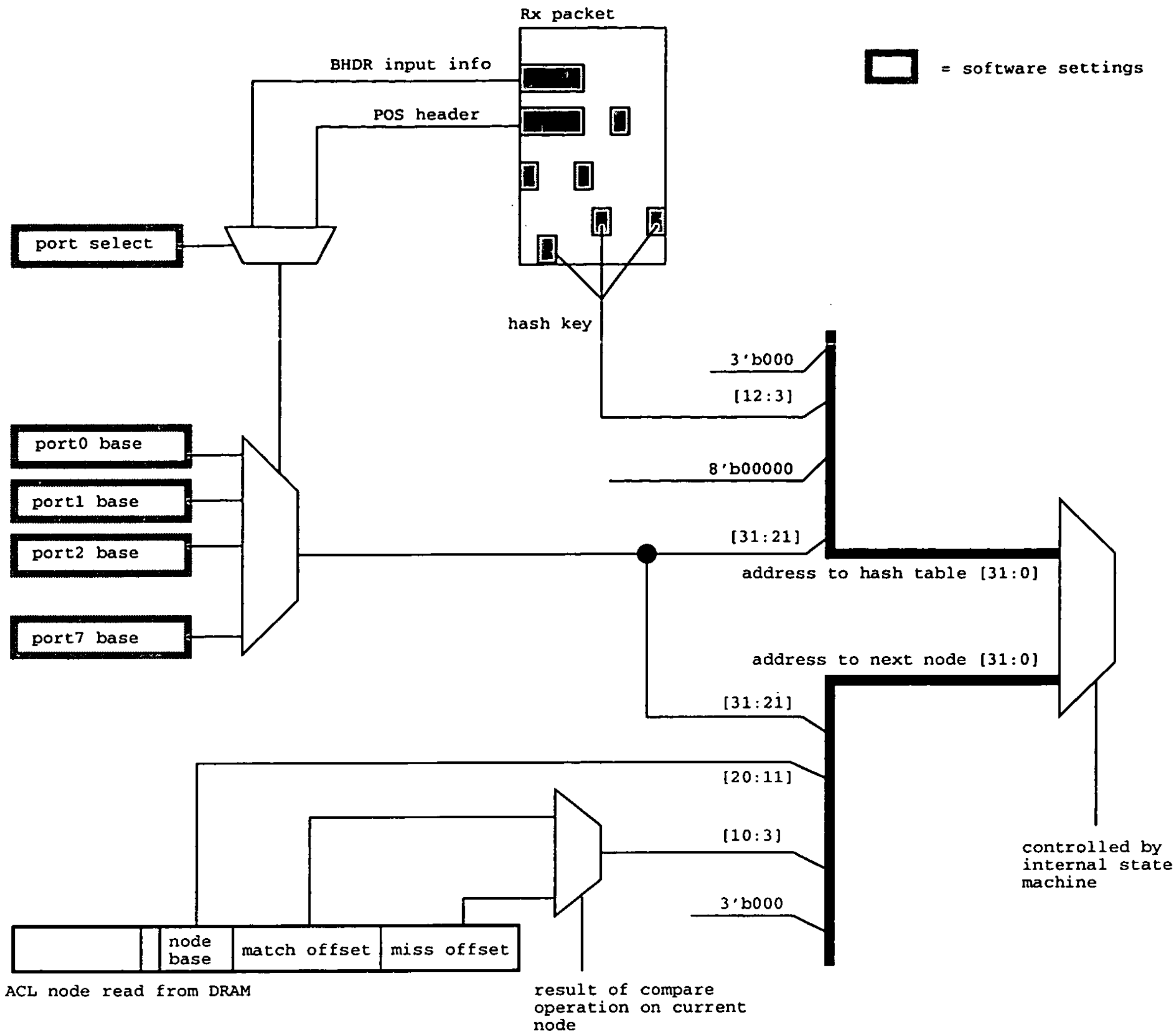


Figure 2: Addressing logic for ACL lookup engine

2.1.7 Expected performance

Performance calculations with Salsa4 are made complicated by the hash table. Some lists off the hash table may be over 70 nodes long, others just 1 node long. The worst, typical and best case numbers below use the longest, average and shortest length lists in each hash table to illustrate this fact. As discussed before, we expect incoming traffic to hit all buckets of the hash table, evenly and therefore, the typical number would be most likely seen.

These calculations are based on a 650Kpps packet rate, with any ACL engine delays added to the 1.538usec per-packet switching delay. Each "check" takes 160ns (includes RAS precharge time on the EDO) to complete.

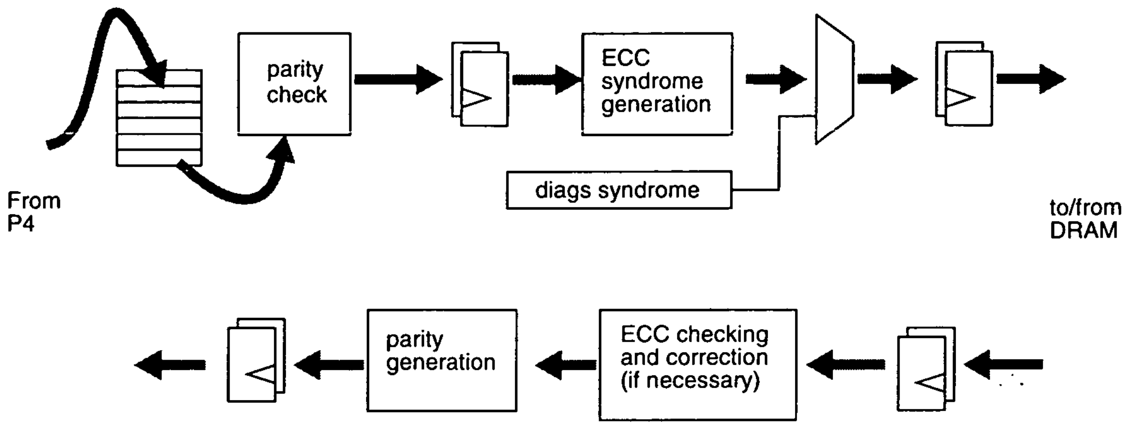
Table 3: Expected Performance of Extended Access Lists with Salsa4's ACL Engine

Access List	Performance Worst	Performance Typical	Performance Best
Internal (256 entries)	28Kpps	266Kpps	650Kpps
Internal (512 entries)	14Kpps	171Kpps	650Kpps
AOL (315 entries)	91Kpps	352Kpps	558Kpps
McGill (91entries)	86Kpps	352Kpps	485Kpps
ATT (322 entries)	192 checks 31Kpps	7 checks 376Kpps	2 checks 538Kpps
BT (1035 entries)	214 checks 28Kpps	24 checks 186Kpps	2 checks 538Kpps
SWBell1 (940 entries)	1768 checks 5Kpps	113 checks 51Kpps	4 checks 460Kpps
SWBell2 (821 entries)	488 checks 12Kpps	10 checks 319Kpps	0 checks 650Kpps

2.2 ECC Protection on DRAM interface

ECC protection on all 64bits is being added between Salsa4 and the EDO DRAM Routing table Memory. The 8 additional bits on this interface currently being used for parity, will be used as part of the ECC syndrome.

The integration of the ECC data path with the existing parity checking path is shown below:.



2.2.1 Parity handling on DRAM writes by P4

During a write, the only check is for parity on data being read out of the write buffer. If a parity error is detected,
- interrupt is sent to the processor
- write address is captured
- appropriate bits are set in the "DRAM Error Status Register"
- write is aborted.

If there is no parity error or if checking is disabled, parity bits are dropped, an ECC syndrome is calculated and written to DRAM. It is possible for software to substitute the Salsa-calculated syndrome with a value written into a register.

2.2.2 Single-bit error on a DRAM read by P4

If ECC-single-bit-error-notification is enabled,
- the processor is interrupted
- DRAM address is captured.
- Corrected data (if single-bit-error-correction is enabled) or original data (if not) is passed on to the parity generation logic and then on to the P4
- the occurrence of this error is captured in a clear-on-read bit in the "ECC Status Register"

if ECC-single-bit-error-notification is disabled:
- the processor is not interrupted
- address and status are still captured
- Corrected data (if single-bit-error-correction is enabled) or original data (if not) is passed on to the parity generation logic and then on to the P4

2.2.3 Multi-bit error on a DRAM read by P4

It is not possible to correct this type of error.

If Bus-error-on-multibit-ECC-error is enabled
- Bad data is forwarded to the P4 with good parity, but with a bus error.
- DRAM address is captured
- the occurrence of this error is captured in a clear-on-read bit in the "ECC Status Register"

If Bus-error-on-multibit-ECC-error is disabled
- Bad data is forwarded to the P4 with good parity, and no bus error
- address and status are still captured

2.2.4 Single/Multi Bit Errors on DRAM reads by ACL/MTRIE engines

When the ACL and/or MTRIE LU engines encounter an ECC error, notification is made through status registers associated with each incoming packet.

If ECC-single-bit-error-notification is enabled,
- a bit is raised in either the "ACL Engine Status Register" or the "Receive BMA Packet Synopsis Register".
- DRAM address is captured.
- Corrected data (if single-bit-error-correction is enabled) or original data (if not) is used in the lookup
- the occurrence of this error is captured in a clear-on-read bit in the "ECC Status Register"
- the engine continues searching

if ECC-single-bit-error-notification is disabled:
- ECC SBE's will not set the error bit in the engine status registers
- address and status are still captured
- Corrected data (if single-bit-error-correction is enabled) or original data (if not) is used in the lookup
- the engine continues searching

If Bus-error-on-multibit-ECC-error is enabled
- a bit is raised in either the "ACL Engine Status Register" or the "Receive BMA Packet Synopsis Register".
- DRAM address is captured
- the occurrence of this error is captured in a clear-on-read bit in the "ECC Status Register"
- the engine stops immediately

If Bus-error-on-multibit-ECC-error is disabled
- ECC MBE's will not set the error bit in the engine status registers
- address and status are still captured
- the engine stops immediately

Section 3. Manufacturing & Test Considerations

The following is the output of lsistats, on the first revision of Salsa. It is packaged in a 503 pin EPBGA.

```
Section One: Design Summary
===========================

Top Module:              salsa
Technology:              lcbg10p
Array Type:              lcbg10p
Mftg Code:               Not Available.
Pad Pitch:               2.9wire
Dimension:               (X = 9.510, Y = 9.510)
Routing Layers:          2
Package Name:            iw51
Package Desc:            Not Available.
Primary Voltage:         None Specified.
Secondary Voltage:       None Specified.


Section Two: Design Statistics Summary
======================================

(1). I/O Statistics

Input Pins Used:         31     Input Pads Used:        29     Input Slots Used:        31
Output Pins Used:        67     Output Pads Used:       1      Output Slots Used:       1
Bidirect Pins Used:      242    Bidirect Pads Used:     308    Bidirect Slots Used:     308


Total Pins Used:         340
Total Pads Used:         338
Total Slots Used:        340


(2). Design Statistics

Logic Units Used (lu):          360655.00
Megacell Units Used (mu):       227083.17
Chip Raw Units:                 3249856.00
Logic Units Usage:              0.119
Chip Usage:                     0.181

Total Cells:                    26592
Total Cell Types:               163
Total Units (lu + mu):          587738.17
Total Signal Nets:              28745
Total NC Nets:                  2
Average Pins/Nets:              3.616
```

An additional 112 slots on the die are used for power/ground pairs.

Section 4. Software Considerations

The following tables show the Address Maps for the linecard, linecard-I/O and Salsa4. There have not been any changes to these maps in this rev of Salsa.

4.1 Line-Card Address Map

0xFFFF_FFFF		
0xF000_0000	Receive Pkt. Mem. (not for TTM)	256M
0xE000_0000	Receive Packet Memory	256M
0xD000_0000	Transmit Pkt Memory (not for TTM)	256M
0xC000_0000	Transmit Packet Memory	256M
0x8000_0000	Reserved	1G
0x5000_0000	Main Memory (Expansion)	768M
0x4000_0000	Main Memory *	256M
0x2000_0000	Reserved	512M
0x1800_0000	Boot FLASH Memory	128M
0x1000_0000	Line-Card I/O Address Space	128M
0x0000_0000	Main Memory *	256M

* Mapped at mupltiple locations

4.2 Line-Card I/O Address Map

0x17FF FFFF		
0x1200 0000	PLIM Address Space	96M
0x11C0 0000	Maintenance Address Space	4M
0x1180 0000	ToFab FIA ASIC Address Space	4M
0x1140 0000	FrFab FIA ASIC Address Space	4M
0x10C0 0000	Receive BMA ASIC Address Space	8M
0x1040 0000	Transmit BMA ASIC Address Space	8M
0x1000 0000	SalsaASIC Address Space	4M

4.3 Salsa ASIC Address Map

0x103F FFFF		
0x1030 0000	Reserved**	1M
0x102C 0000	Reserved**	256K
0x1028 0000	Transmit Packet Window	256K
0x1024 0000	Reserved**	256K
0x1020 0000	Receive Packet Window	256K

		1M
0x1010 0000	Reserved **	
		1M
0x1000 0000	Salsa Register Address Space	

** These areas are reserved for future expansion. For this implementation version of the Salsa ASIC, accesses to these address spaces are illegal and therefore result as bus-error for read operations or erroneous interrupt for write operations

Section 5. Updated Register List

For the sake of completeness, both existing and new registers are shown below. Existing registers are shaded.

Table 4: Summary of Registers

Register Name	Access Type	Address	Section
Watch-Dog Timer	16-bit R/W	0x1000 0004	1
General-Purpose Counter	16-bit R/W	0x1000 000C	2
Real-Time Interrupt Timer	16-bit R/W	0x1000 0014	3
Receive Network Disable Timer	16-bit R/W	0x1000 001C	4
Transmit Network Disable Timer	16-bit R/W	0x1000 0024	5
Receive BMA Bus Time-Out Counter	16-bit R/W	0x1000 002C	6
Transmit BMA Bus Time-Out Counter	16-bit R/W	0x1000 0034	7
Line-card I/O Bus Time-Out Counter	16-bit R/W	0x1000 003C	8
Timers/Counters Control Register	8-bit R/W	0x1000 0044	9
Interrupt Cause Register	16-bit R	0x1000 004C	10
Interrupt Mask Register	16-bit R/W	0x1000 0054	11
Real-Time Interrupt Clear Register	8-bit W	0x1000 005C	12
Reset to BMA Register	8-bit R/W	0x1000 0064	13
Receive Packet Done Register	8-bit W	0x1000 006C	14
Transmit Packet Done Register	8-bit W	0x1000 0074	15
L3 Asic ID Register	16-bit R	0x1000 007C	16
Memory Configuration Register	16-bit R/W	0x1000 0084	17
Memory Combination Register	8-bit R/W	0x1000 008C	18
L3 Performance Enhancment Register	8-bit R/W	0x1000 0094,	19
Error Checking Enable Register	8-bit R/W	0x1000 009C,	20
Receive BMA Bus Error Status Register	8-bit R	0x1000 00A4,	21
Transmit BMA Bus Error Status Register	8-bit R	0x1000 00AC,	22
DRAM Error Status Register	8-bit R	0x1000 00B4	23
Line-card I/O Bus Error Status Register	8-bit R	0x1000 00BC	24
DRAM Catastrophic Error Address Register	32-bit R	0x1000 00C4	25

Table 4: Summary of Registers

Register Name	Access Type	Address	Section
Receive BMA Address Exception Register	32-bit R	0x1000 00CC	26
Transmit BMA Address Exception Register	32-bit R	0x1000 00D4	27
I/O Address Exception Register	32-bit R	0x1000 00DC	28
L3 Interrupt Status Information Register	8-bit R	0x1000 00E4	29
Receive BMA Packet Synopsis Register	16-bit R/W	0x1000 00F4	30
Receive BMA Hardware Assist Register	8-Bit R/W	0x1000 00FC	31
Receive BMA Buffer Service Information Register	8-bit R	0x1000 0104	32
Receive BMA Buffer Flush Information Register	8-bit R	0x1000 010C	33
Transmit BMA Buffer Service Information Register	8-bit R	0x1000 0114	34
Transmit BMA Buffer Flush Information Register	8-bit R	0x1000 011C	35
Receive BMA Packet Updated Info Register	32-bit R	0x1000 0124	36
Receive BMA Protocol 0 Identifier Register	32-bit R/W	0x1000 012C	37
Receive BMA Protocol 1 Identifier Register	32-bit R/W	0x1000 0134	38
Receive BMA Protocol 2 Identifier Register	32-bit R/W	0x1000 013C	39
Receive BMA Protocol 3 Identifier Register	32-bit R/W	0x1000 0144	40
FIB Root Regsiter	32-bit R/W	0x1000 014C	41
Leaf Pointer Register	32-bit R	0x1000 0154	42
Lookup Result Register	32-bit R	0x1000 015C	43
Diagnostic access to Receive BMA Prefetch Buffer 0	64-bit R/W (byte WR)	1000_1000 - 1000_105F	
Diagnostic access to Receive BMA Prefetch Buffer 1	64-bit R/W (byte WR)	1000_1080 - 1000_10DF	
Diagnostic access to Receive BMA Prefetch Buffer 2	64-bit R/W (byte WR)	1000_1100 - 1000_115F	
Diagnostic access to Transmit BMA Prefetch Buffer 0	64-bit R/W (byte WR)	1000_1200 - 1000_125F	

Table 4: Summary of Registers

Register Name	Access Type	Address	Section
Diagnostic access to Transmit BMA Prefetch Buffer 1	64-bit R/W (byte WR)	1000_1280 - 1000_12DF	
Diagnostic access to Transmit BMA Prefetch Buffer 2	64-bit R/W (byte WR)	1000_1300 - 1000_135F	
Diagnostic access to DRAM Write Buffer	64-bit R/W	1000_1500 - 1000_15BF	
Diagnostic access to Receive BMA Write Buffer	64-bit R/W	1000_1600 - 1000_16BF	
Diagnostic access to Transmit BMA Write Buffer	64-bit R/W	1000_1700 - 1000_17BF	
Port0 ACL Tree Base Register	32bit R/W	0x1000 0164	44
Port1 ACL Tree Base Register	32bit R/W	0x1000 016C	45
Port2 ACL Tree Base Register	32bit R/W	0x1000 0174	46
Port3 ACL Tree Base Register	32bit R/W	0x1000 017C	47
Port4 ACL Tree Base Register	32bit R/W	0x1000 0184	48
Port5 ACL Tree Base Register	32bit R/W	0x1000 018C	49
Port6 ACL Tree Base Register	32bit R/W	0x1000 0194	50
Port7 ACL Tree Base Register	32bit R/W	0x1000 019C	51
ACL Engine Config Register	16bit R/W	0x1000 01A4	52
Current ACL Node HI Bytes Register	32bit R	0x1000 01AC	53
Current ACL Node LO Bytes Register	32bit R	0x1000 01B4	54
Current ACL Hash Key Register	8bit R	0x1000 01BC	55
ACL Engine Status Register	32bit R	0x1000 01C4	56
ACL Hash Key Selection Register	32bit R/W	0x1000 01CC	57
ACL Hash Key Selection Register	32bit R/W	0x1000 01D4	58
ACL Hash Key Selection Register	16bit R/W	0x1000 01DC	59
ACL Compare Mask 0 Register	32bit R/W	0x1000 01E4	60
ACL Compare Mask 1 Register	32bit R/W	0x1000 01EC	61
ACL Compare Mask 2 Register	32bit R/W	0x1000 01F4	62
ACL Compare Mask 3 Register	32bit R/W	0x1000 01FC	63
ACL Compare Mask 4 Register	32bit R/W	0x1000 0204	64
ACL Compare Mask 5 Register	32bit R/W	0x1000 020C	65

Table 4: Summary of Registers

Register Name	Access Type	Address	Section
ACL Compare Mask 6 Register	32bit R/W	0x1000 0214	66
ACL Compare Mask 7 Register	32bit R/W	0x1000 021C	67
ACL Compare Mask 8 Register	32bit R/W	0x1000 0224	68
ACL Compare Mask 9 Register	32bit R/W	0x1000 022C	69
ACL Max Nodes Register	16bit R/W	0x1000 0234	70
ACL Current Count Register	16bit R	0x1000 023C	71
ECC Status Register	16bit R	0x1000 0244	72
ECC Diagnostic Syndrome Register	8bit R/W	0x1000 024C	73
ECC Config Register	8bit R/W	0x1000 0254	74
ECC SBE Address Register	32bit R	0x1000 025C	75
ECC SBE Syndrome Register	8bit R	0x1000 0264	76
ECC MBE Syndrome Register	8bit R	0x1000 026C	77

There are eight count-down counters implemented in the Salsa ASIC to serve various purposes. All of these counters are 16-bit read/write. On power-on reset, all counters are disabled and loaded with a default count of 65,535 (0xFFFF). Under software control, each counter can be loaded with a known value and enabled through proper write operations. Each timer will count as many usec clocks as the programmed value (i.e. a programmed value of 5, will produce a timed interval of 4 to 5 usecs). All reads & writes to the counters must be 16-bit accesses.

5.1 Watch-Dog Timer

16-bit R/W
0x1000 0004

After reset, the timer is disabled and loaded with a default value of 0xFFFF, i.e. 65.535 msecs.
Start values other than the default 0xFFFF, can be programmed by writing to this address. If enabled the timer will automatically decrement, every 1us if there is a non-zero value in the counter. The enable bit can be found in the "Timers/Counters Control Register" on page 24.

Should the timer reach 0x0000, an NMI interrupt will be generated and the timer stops counting. To clear the NMI flag, software must program a new value to the Watch Dog timer. A value of 0x0000 is acceptable if it is desired not to restart the timer.

At any time, the Watch Dog Timer Enable may be switched off, At that point, any value can be written to the Watch Dog timer, without triggering the decrement logic. Note: This timer will not generate a reset at

end of count, as previously proposed.

5.2 General-Purpose Counter

16-bit R/W
0x1000 000C

After reset, the counter is disabled and loaded with a default value of 0xFFFF.

The counter is enabled and disabled through the "Timers/Counters Control Register" on page 24. A predetermined count-down value can be loaded with a 16-bit write operation at this address. Otherwise, an after-reset default value of 0xFFFF will be assumed.

Should the timer reach 0x0000, an interrupt will be generated and the timer stops counting. To clear the interrupt flag, software must program a new value to the General Purpose counter. A value of 0x0000 is acceptable if it is desired not to restart the timer. At any time, the General Purpose counter Enable may be switched off, Once this is done, any value can be written to the timer, without triggering the decrement logic. *Note: This is different from the IRSP interrupt scheme. Software does not need to keep polling this counter!*

5.3 Real-Time Interrupt Timer

16-bit R/W
0x1000 0014

After reset, the timer is disabled and loaded with a default value of 0xFFFF.

The timer is enabled and disabled through the "Timers/Counters Control Register" on page 24. A predetermined count can be loaded with a 16-bit write operation at this address. Otherwise, an after-reset default value of 0xFFFF will be assumed.

If the counter is enabled, a decrement occurs once every usec. It will count down until it is disabled or when it reaches a value of 0x0001. As a result, an interrupt will be latched and issued. On the next clock the timer will be reloaded with the previous loaded value. Software needs to acknowledge the interrupt and clear the interrupt latch by performing a write operation to the "Real-Time Interrupt Clear Register" on page 26.

5.4 Receive Network Disable Timer

16-bit R/W
0x1000 001C

After reset, the timer is disabled and loaded with a default value of 0x0000.

This special timer is used to disable the *receive network interrupt* for a period of time. Software must write a 16-bit value to this register, causing it to start decrementing once every 20ns if the timer is enabled. During the decrement, the receive network interrupt will be blocked. When the timer value reaches zero, it will remain there and un-block the receive network interrupt. The timer will not be activated until software

writes to it again. Note: This timer does not generate an interrupt!

5.5 Transmit Network Disable Timer

16-bit R/W
0x1000 0024

After reset, the timer is disabled and loaded with a default value of 0x0000.

This special timer is used to disable the transmit network interrupt for a period of time. Software must write a 16-bit value to this register, causing it to start decrementing once every 20ns if the timer is enabled. During the decrement, the transmit network interrupt will be blocked. When the timer value reaches zero, it will remain there and un-block the transmit network interrupt. The timer will not be activated until software writes to it again. Note: This timer does not generate an interrupt!

5.6 Receive BMA Bus Time-Out Counter

16-bit R/W
0x1000 002C

After reset, the timer is disabled and loaded with a default value of 0xFFFF.

A predetermined count can be loaded with a 16-bit write operation at the timer's address. The timer, however, will not be activated until a legal operation is initiated on the Receive BMA Bus by the L3 ASIC. The timer will be reloaded with the programmed count value and start counting down at a rate of one microseconds immediately after the Request cycle. It will stop counting if the counter is exhausted or when the bus transaction ends, i.e. when the **bma_req_l** goes away. If the timer expires before the end of the bus transaction, a corresponding error bit will be set in the Receive BMA Bus Error Status Register. Depending on the operation type, a bus error or an erroneous interrupt is posted to the P4 processor. *Note: The count value should be selected to be larger than the longest bus transaction, i.e. the prefetch operation.*

5.7 Transmit BMA Bus Time-Out Counter

16-bit R/W
0x1000 0034

After reset, the timer is disabled and loaded with a default value of 0xFFFF.

A predetermined count can be loaded with a 16-bit write operation at the timer's address. The timer, however, will not be activated until a legal operation is initiated on the Transmit BMA Bus by the L3 ASIC. The timer will be reloaded with the programmed count value and start counting down at a rate of one microseconds immediately after the Request cycle. It will stop counting if the counter is exhausted or when the bus transaction ends, i.e. when the **bma_req_l** goes away. If the timer expires before the end of the bus transaction, a corresponding error bit will be set in the Transmit BMA Bus Error Status Register. Depending on the operation type, a bus error or an erroneous interrupt is posted to the P4 processor. *Note: The count value should be selected to be larger than the longest bus transaction, i.e. the prefetch operation.*

5.8 Line-card I/O Bus Time-Out Counter

16-bit R/W
0x1000 003C

After reset, the timer is disabled and loaded with a default value of 0xFFFF.

The timer is enabled or disabled through its enable bit in the Counter Control Register @0x1000 0044. A predetermined count can be loaded with a 16-bit write operation at the timer's address. The timer, however, will not be activated until a legal operation is initiated on the Line-Card I/O Bus by the L3 ASIC. The timer will be reloaded with the programmed count value and start counting down at a rate of one microseconds immediately after the Request cycle. It will stop counting if the counter is exhausted or when the bus transaction ends, i.e. when the **io_strobe_l** goes away. If the timer expires before the end of the bus transaction, a corresponding error bit will be set in the Line-Card I/O Bus Error Status Register. Depending on the operation type, a bus error or an erroneous interrupt is posted to the P4 processor. Note: The count value should be selected to be larger than the longest bus transaction.

5.9 Timers/Counters Control Register

8-bit R/W
0x1000 0044

After reset, the register is loaded with a default value of 0x00.

The register is read/write through 8-bit operations. Immediately after reset, the P4 processor needs to write to this register with bit 2 set to "1" in order to enable the real-time interrupt.

Bit	Description
4	Watch Dog Timer Enable 0 - Disabled 1- Enable
3	General-Purpose Counter Enable 0 - Disabled 1 - Enabled
2	Real-Time Timer Enable 0 - Disabled 1 - Enabled
1	"Receive Network Disable" Timer Enable 0 - Disabled 1 - Enabled
0	"Transmit Network Disable" Timer Enable 0 - Disabled 1 - Enabled

5.10 Interrupt Cause Register

16bit R
0x1000 004C

After reset, the register is loaded with a default value of 0x0000.

This register captures the source of the general interrupt, sent to the P4.

Bit		Description
[15:14]		Reserved.
13	Indirect 8	Receive PSA ASIC (external)
12	Direct #0	Receive Network Interrupt.
11	Direct #1	Transmit Network Interrupt.
10	Direct #2	Maintenance Bus Interrupt.
9	Direct #3	L3 Interrupt.
8	Indirect #0	Receive BMA ASIC (external).
7	Indirect #1	Transmit BMA ASIC (external).
6	Indirect #2	ToFab FIA ASIC (external).
5	Indirect #3	FrFab FIA ASIC (external)
4	Indirect #4.	PLIM Interrupt Int-0 (external)
3	Indirect #5.	PLIM Interrupt Int-1 (external)
2	Indirect #6a	General Purpose Counter Interrupt.
1	Indirect #6b	Real-Timer Interrupt.
0	Indirect #7	Error Interrupt.

5.11 Interrupt Mask Register

16bit R/W
0x1000 0054

After reset, the register is loaded with a default value of 0xFFFF, i.e. masking all direct and indirect interrupt sources.

The register is read/write through a 16-bit operation. Software can individually enable any interrupts by writing a "0" into the corresponding bits of the register.

Bit	Description
[15:13]	Reserved.
12	Reveive PSA ASIC Interrupt Mask Bit
11	Receive Network Interrupt Mask Bit
10	Transmit Network Interrupt Mask Bit
9	Maintenance Bus Interrupt Mask Bit

Bit	Description
8	L3 Interrupt Mask Bit
7	Receive BMA ASIC Interrupt Mask Bit
6	Transmit BMA ASIC Interrupt Mask Bit
5	ToFab FIA ASIC Interrupt Mask Bit
4	FrFab FIA ASIC Interrupt Mask Bit
3	PLIM Interrupt-0 Mast Bit
2	PLIM Interrupt-1 Mask Bit
1	Timer Interrupt Mask Bit (masks both General Purpose and Real Time, timer interrupts
0	Error Interrupt Mask Bit

5.12 Real-Time Interrupt Clear Register

8 bit W
0x1000 005C

An 8-bit write operation to this register will clear the real-time interrupt. Data pattern is don't care.

5.13 Reset to BMA Register

8 bit R/W
0x1000 0064

After reset, the register is loaded with a default value of 0x0000.

This register is used to send a reset to the BMA logic. A level reset signal is generated from the value stored in each bit. This means that each reset must be turned on and off, by setting and clearing the corresponding bit.

Bit	Description
[7:2]	Reserved.
1	Rx BMA reset 0 = No reset 1 = Initiates a reset to the Rx BMA asic
0	Tx BMA reset 0 = No reset 1 = Initiates a reset to the Tx BMA asic

5.14 Receive Packet Done Register

8-bit W
0x1000 006C

A write operation performed on this register will clear the Receive Network Service Request bit of the in-service prefetch buffer. This in turn will set the corresponding Flush Request bit for this selected prefetch buffer. In order to preserve the P4's order, the flush request is queued into the Receive BMA write buffer. This ensures that all the previous write requests from the P4 processor will be executed before this flush request.

5.15 Transmit Packet Done Register

8 bit W
0x1000 0074

A write operation performed on this register will clear the Transmit Network Service Request bit of the in-service prefetch buffer. This in turn will set the corresponding Flush Request bit for this prefetch buffer. In order to preserve the 7P4's order, the flush request is queued into the Transmit BMA write buffer. This ensures that all the previous write requests from the P4 processor will be executed before this flush request.

5.16 L3 Asic ID Register

16 bit R
0x1000 007C

This is a read-only register. Sixteen bits of information are extracted from the JTAG ID register which has the following format. The register is always read as 16'h10C9.

Bit	Description
15:12	ASIC Revision.
11:0	ASIC Part Number.

5.17 Memory Configuration Register

16-bit R/W
0x1000 0084

After reset, the register is loaded with a default value of 026E, i.e. 2 clock CAS precharge, 5 clock RAS precharge, 4 clock WRITE tRCD, 5 clock READ tRCD, refresh rate = 15 uS.

The values on power-up are suited for a 60ns EDO. For a 50ns EDO, this register should be programmed with a value of 000E, i.e. 2 clock CAS precharge, 4 clock RAS precharge, 4 clock read and write tRCD, refresh rate = 15uS. The feature to increase tCP to 3 clocks is intended for lab use, and will cause performance degradation. In order to use this feature, tRCD must be at least 4 clocks.

Bit	Description
15:11	Reserved.
10	Clocks spent in CAS precharge, tCP, following an access. Applies to single and burst accesses. 0: 2 clock 1: 3 clocks
9:8	Clocks spent in RAS precharge (i.e RAS HI) 00: 3 clocks 01: 4 clocks 10: 5 clocks 11: 6 clocks
7:6	Clocks spent during tRCD (i.e RAS-to-CAS) on write accesses 00: 3 clocks 01: 4 clocks 10: 5 clocks 11: 6 clocks
5:4	Clocks spent during tRCD (i.e RAS-to-CAS) on read accesses 00: 3 clocks 01: 4 clocks 10: 5 clocks 11: 6 clocks
3:0	DRAM Refresh Timer Value (in microseconds) 1111 - Refresh every 16 microseconds 1110 - Refresh every 15 microseconds : 0000 - Refresh every microsecond

5.18 Memory Combination Register

8-bit R/W
0x1000 008C

After reset, the register is loaded with a default value of 00x11, i.e. Single DIMM with 16Mbytes

Bit	Description
7:6	Reserved
5	Number of DIMMs being used 0 - Single DIMM. 1 - Two DIMM..
4:0	DIMM combination must be one of

[4:0]	Dimm 0	Dimm 1
00000	8	8 or noth-
00001	16	8
00010	32	8
00011	64	8
00100	128	8
00101	256	8
N/A	512	8
00110	16	16 or
00111	32	16
01000	64	16
01001	128	16
01010	256	16
01011	512	16
01100	32	32 or
01101	64	32
01110	128	32
01111	256	32
10000	512	32
10001	64	64 or
10010	128	64
10011	256	64
10100	512	64
10101	128	128 or
10110	256	128
10111	512	128
11000	256	256 or

[4:0]	Dimm 0	Dimm 1
11001	512	256
11010	512	512

5.19 L3 Performance Enhancment Register

8-bit R/W
0x1000 0094,

After reset, this register is loaded with a default value of 0x0000, i.e. all read-around-writes and prefetch buffers are disabled.

Bit	Description
7	Reserved.
6	Receive BMA Read-Around-Write Enable 0 - Disabled 1 - Enabled
5	Transmit BMA Read-Around-Write Enable 0 - Disabled 1 - Enabled
4	Main Memory Read-Around-Write Enable 0 - Disabled 1 - Enabled
3	Enable early clear of receive net work interrupt 0 - Disabled 1 - Enabled
2	Enable early clear of transmit network interrupt 0 - Disabled 1 - Enabled
[1:0]	Reserved

5.20 Error Checking Enable Register

8-bit R/W
0x1000 009C,

After reset, this register is loaded with a default value of 0x00, i.e. all checking is disabled.

Bit	Description
[7:5]	Reserved.
4	P4 Timeout Error Checking 0 - Disable P4 timeout checking on reads taking place on the P4 bus 1 - Enable P4 timeout checking on reads taking place on the P4 bus
3	Main Memory Parity Checking & ECC Bus Error Enable 0 - Disable write data parity check by P4 for main memory data. Also Disables Bus Error during reads that show a multibit ECC error 1 - Enable read/write parity check by P4 for main memory data. Also enables Bus Errors on reads that have multi-bit ECC errors in them.
2	Receive BMA Error Checking 0 - Disable error check on the receivte BMA bus interface 1 - Enable error check on the receive BMA bus interface
1	Transmit BMA Error Checking 0 - Disable error check on the transmit BMA bus interface 1 - Enable error check on the transmit BMA bus interface
0	I/O Checking Enable 0 - Disable illegal Address/Data width checking on I/O interface 1 - Enable illegal Address/Data width checking on I/O interface

5.21 Receive BMA Bus Error Status Register

8-bit R
0x1000 00A4,

After reset, this register is loaded with a default value of 0x00, i.e. no error has been detected.

When a Receive BMA bus error occurs during an access, a corresponding bit will be set in this register. If the error had been due to a write operation (as indicated by bit[6] being set), an interrupt to the P4 processor will be generated. If the error had been due to a read operation, a bus-error will be returned to P4. A read from this register will clear all the bits. For all bits, a "1" value indicates an Error of the corresponding type has occurred.

Bit	Description

Bit	Description
10	Receive BMA Interrupt errors: Logical OR of bits [7], [6], [5], [2], [1] and [0].
9	Receive BMA Non-interrupt errors: Logical OR of bits [8], [4] and [3].
8	Timeout error on BMA bus, during Read
7	Timeout error on BMA bus, during Packet prefetch
6	Timeout error on BMA bus, durng Write dispatch
5	Timeout error on BMA bus, during Packet flush.
4	Bus Read Parity Error
3	Packet service (read from prefetched packet) parity error
2	Packet prefetch from Receive BMA, parity error
1	Write dispatich parity error
0	Packet Flush parity error

5.22 Transmit BMA Bus Error Status Register

0x1000 00AC,
8-bit R

After reset, this register is loaded with a default value of 0x00, i.e. no error has been detected.

When a Transmit BMA bus error occurs during an access, a corresponding bit will be set in this register. If the error had been due to a write operation (as indicated by bit[6] being set), an interrupt to the P4 processor will be generated. If the error had been due to a read operation, a bus-error will be returned to P4. A read from this register will clear all the bits. For all bits, a "1" value indicates an Error of the corresponding type has occurred.

Bit	Description
10	Transmit BMA Interrupt errors: Logical OR of bits [7], [6], [5], [2], [1] and [0].
9	Transmit BMA Non-interrupt errors: Logical OR of bits [8], [4] and [3].
8	Timeout error on BMA bus, during Read
7	Timeout error on BMA bus, during Packet prefetch
6	Timeout error on BMA bus, durng Write dispatch
5	Timeout error on BMA bus, during Packet flush.
4	Bus Read Parity Error
3	Packet service (read from prefetched packet) parity error

Bit	Description
2	Packet prefetch from Receive BMA, parity error
1	Write dispatich parity error
0	Packet Flush parity error

5.23 DRAM Error Status Register

0x1000 00B4
8-bit R

After reset, this register is loaded with a default value of 0x00, i.e. no error has been detected.

When a DRAM error occurs during an access, a corresponding bit will be set in this register. If the error had been due to a write operation (as indicated by bit[4] being set), an interrupt to the P4 processor will be generated. If the error had been due to a read operation, a bus-error will be returned to P4. A read from this register will clear all the bits.

A special note about partial writes: P4 partial writes are handled as read-modify-write operations in Salsa4. Should an uncorrectable ECC error (MBE) occur during this operation, the write is aborted and both write and read error bits are set. This register's value becomes 33 during that time.

Bit	Description
[7:6]	Reserved
5	DRAM Read Error (OR condition of bits [0] and [3], shown below) 0 - No error 1 - Error
4	DRAM Write Error (OR condition of bits [1-2], shown below) 0 - No error 1 - Error
3.	DRAM Read Address Error 0 - No Error 1 - Read address is out-of-bounds for current memory combination specified in Memory Combination Register, page 29
2	DRAM Write Address Error 0 - No Error 1 - Write address is out-of-bounds for current memory combination specified in Memory Combination Register, page 29
1	DRAM Write Parity Error 0 - No Error

Bit	Description
	1 - Error
0	DRAM Read ECC Single/Multibit Error.
	0 - No error
	1 - Either a multibit or single bit error has occurred

5.24 Line-card I/O Bus Error Status Register

0x1000 00BC
8-bit R

After reset, this register is loaded with a default value of 0x00, i.e. no error has been detected. When an I/O bus error occurs during an access, a corresponding bit will be set in this register. If the error had been due to a write operation (as indicated by bit[4] being set), an interrupt to the P4 processor will be generated. If the error had been due to a read operation, a bus-error will be returned to P4. A read from this register will clear all the bits.

Bit	Description
[7:6]	Reserved.
5	I/O Bus Read Error (OR Condition of bits [1] and [3], shown below)
	0 - No error
	1 - Error
4	I/O Bus Write Error (OR Condition of bits [0] and [2], shown below)
	0 - No error
	1 - Error
3	Line-card I/O Bus Read Time-Out Error.
	0 - No error
	1 - A bus timeout occurred before io_rdy_l was asserted
2	Line-card I/O Bus Write Time-Out Error.
	0 - No error
	1 - A bus timeout occurred before io_rdy_l was asserted
1	Illegal Line-card I/O Read Accessing Error
	0 - No error
	1 - An invalid read operation is attempted by P4. Please see the summary of illegal I/O bus operatons listed in Chapter 3 of the BFR Quad OC3 Linecard spec. (ENG-7439).

Bit	Description
0	Illegal Line-card I/O Write Accessing Error 0 - No error 1 - An invalid write operation is attempted by P4. Please see the summary of

5.25 DRAM Catastrophic Error Address Register

0x1000 00C4
32-bit R

After reset, the register is cleared, i.e. being loaded with a default value of 0x00.

This register captures the 32bit address at the first occurrence of any error (since its last reading) of the "DRAM Error Status Register" on page 33, except for SBE's. A separate address register for single-bit-errors can be found in "ECC SBE Address Register" on page 52.

Bit	Description
[31:0]	Address Bits [31:0]

5.26 Receive BMA Address Exception Register

0x1000 00CC
32-bit R

After reset, the register is cleared, i.e. being loaded with a default value of 0x00.

This register will capture 32 bits of the address at which the error occured

Bit	Description
[31:0]	Address Bits [31:0]

5.27 Transmit BMA Address Exception Register

0x1000 00D4
32-bit R

After reset, the register is cleared, i.e. being loaded with a default value of 0x00.

This register will capture 32 bits of the address at which the error occured.

Bit	Description
[31:0]	Address Bits [31:0]

5.28 I/O Address Exception Register

0x1000 00DC
32-bit R

After reset, the register is cleared, i.e. being loaded with a default value of 0x00.
This register will capture the 32 bits of the address at which the error occured.

Bit	Description
[31:0]	Address Bits [31:0]

5.29 L3 Interrupt Status Information Register

0x1000 00E4
8-bit R

This register further breaks down the cause of the L3 Error Interrupt bit, bit 0[, in "Interrupt Cause Register" on page 25. All causes listed here are from internal L3 Error checking and timer logic.

Bit	Description
[7:5]	Reserved
4	rx_bma_error_int
3	tx_bma_error_int
2	dram_error_int
1	io_error_int
0	General Purpose or Real-Time timers have expired

5.30 Receive BMA Packet Synopsis Register

16bit R/W
0x1000 00F4

After reset, the value of this register is all zeros.

Only bit[4] of this register is write accessible. Any upper bits are ignored during a P4 write operation.

Bit	Description
15:11	Reserved

Bit	Description
10	Copy of THB bit from BHDR
9	0 - Packet passed length check 1 - Error: Packet failed length check
8	0 - Packet is IPv4, no options 1 - Error: This is not a "fast-path" packet
7	0 - Protocol identifier in MAC matches matches expected value 1 - Error: Unknown protocol identifier in MAC
6	0 - Prior to decrement TTL value was larger than 1 1 - Error: Packet TTL is 1 or 0
5	0 - Checksum validation passed 1 - Error: Incorrect checksum detected
4	0 - Disable hardware generated TTL and Checksum updates when flushing the current packet 1 - Okay to use hardware generated TTL and Checksum when flushing the current packet Note, this disable automatically clears with each new packet.
3	A sampling of the MTRIE lookup enable bit in register "Receive BMA Hardware Assist Register" on page 37, at the time the lookup starts.
2	0 - No out-of-range error during lookup reads 1 - Error: Encountered an out-of-range read address during lookup read
1	0 - No parity error 1 - Error: A parity mismatch occurred during MTRIE lookup
0	0 - MTRIE lookup completed successfully 1 - Error: Could not complete MTRIE lookup because no leaf was found after using all octets of IP destination address

5.31 Receive BMA Hardware Assist Register

8-Bit R/W
0x1000 00FC

After reset, this register contains the value 04.

This register enables/disables MTRIE lookup and/or hardware TTL decrement and Checksum updates for all packets. Please note that bit[4] of the "Receive BMA Packet Synopsis Register" disables only on a per-packet basis whereas bit[0] here, disables TTL and checksum updates for all packets.

Bit	Description
7:3	Reserved

Bit	Description
2	0 - Disable 1 - Enable 64 byte reversed prefetch from BMA. The value of Bit[1] is ignored if this bit is 0, and MTRIE lookup is disabled.
1	0 - Disable 1 - Enable MTRIE lookup by hardware
0	0 - Disable 1 - Enable hardware TTL decrement and Checksum recalculation

5.32 Receive BMA Buffer Service Information Register

8-bit R
0x1000 0104

After reset, the register is cleared, i.e. being loaded with a default value of 0x00.

For this buffer there are two associated flags: the Valid Service Request bit and the Flush Request bit. The Valid Service Request bit is set when a buffer is completely filled with the prefetched packet information from the corresponding BMA logic and ready to be processed by the P4 processor. The Flush Request bit of a buffer is set when the microcode is done with the currently selected buffer and wants to have its of data flushed back to the corresponding BMA logic. A buffer is available only when its associated Valid Service Request and Flush Request flags are not set, i.e. "0".

Bit	Description
[7]	Reserved.
[6:5]	Prefetch Buffer Selector 00 = Buffer#0 is currently being selected for prefetching. 01 = Buffer#1 is currently being selected for prefetching. 10 = Buffer#2 is currently being selected for prefetching. 11 = No buffer is selected for prefetching. (equivalent to having the prefetching buffers disabled.
4	Valid Service Request flag for Buffer#2 0 - No service needed 1 - Has packet information data, needs service
3	Valid Service Request flag for Buffer#1 0 - No service needed 1 - Has packet information data, needs service

Bit	Description
2	Valid Service Request flag for Buffer#0 0 - No service needed 1 - Has packet information data, needs service
[1:0]	In-service Buffer Selector 00 = Buffer#0 is currently being selected for in-service process. 01 = Buffer#1 is currently being selected for in-service process. 10 = Buffer#2 is currently being selected for in-service process. 11 = No buffer is selected for in-service. Note: When the selector is set to "11", i.e. no buffer is selected for in-service, it is equivalent to having the in-service buffers disabled

The Received Network Interrupt is asserted when any of the Valid Service Request flags are set. The Valid Service Request flag for the selected in-service buffer will be cleared when the P4 writes to the Packet Done register.

5.33 Receive BMA Buffer Flush Information Register

8-bit R

0x1000 010C

After reset, the register is cleared, i.e. being loaded with a default value of 0x00.

Bit	Description
[7:5]	Reserved.
[4:3]	Flush Buffer Selector 00 = Buffer#0 is currently being selected for flushing. 01 = Buffer#1 is currently being selected for flushing. 10 = Buffer#2 is currently being selected for flushing. 11 = No buffer is selected for flushing.
2	Flush Request flag for Flush Buffer #2. 0 = Inactive 1 = Has data, needs to be flushed back to the received BMA logic.
1	Flush Request Bit for Prefetch Buffer #1. 0 = Inactive

Bit	Description
	1 = Has data, needs to be flushed back to the received BMA logic.
0	Flush Service Request Bit for Prefetch Buffer #0.
	0 = Inactive

5.34 Transmit BMA Buffer Service Information Register

8-bit R
0x1000 0114

After reset, the register is cleared, i.e. being loaded with a default value of 0x00.

Bit	Description
[7]	Reserved.
[6:5]	Prefetch Buffer Selector
	00 = Buffer#0 is currently being selected for prefetching.
	01 = Buffer#1 is currently being selected for prefetching.
	10 = Buffer#2 is currently being selected for prefetching.
	11 = No buffer is selected for prefetching., equivalent to having the prefetching buffers disabled.
4	Valid Service Request flag for Buffer#2
	0 - No service needed
	1 - Has packet information data, needs service
3	Valid Service Request flag for Buffer#1
	0 - No service needed
	1 - Has packet information data, needs service
2	Valid Service Request flag for Buffer#0
	0 - No service needed
	1 - Has packet information data, needs service
[1:0]	In-service Buffer Selector
	00 = Buffer#0 is currently being selected for in-service process.
	01 = Buffer#1 is currently being selected for in-service process.
	10 = Buffer#2 is currently being selected for in-service process.

Bit	Description
	11 = No buffer is selected for in-service. When the selector is set to "11", i.e. no buffer is selected for in-service, it is equivalent to having the in-service buffers disabled.

The Transmitted Network Interrupt is asserted when any of the Valid Service Request flags are set. The Valid Service Request flag for the selected in-service buffer will be cleared when the P4 writes to the Packet Done register.

5.35 Transmit BMA Buffer Flush Information Register

8-bit R
0x1000 011C

After reset, the register is cleared, i.e. being loaded with a default value of 0x00.

Bit	Description
[7:5]	Reserved.
[4:3]	Flush Buffer Selector 00 = Buffer#0 is currently being selected for flushing. 01 = Buffer#1 is currently being selected for flushing. 10 = Buffer#2 is currently being selected for flushing. 11 = No buffer is selected for flushing.
2	Flush Request flag for Flush Buffer #2. 0 = Inactive 1 = Has data, needs to be flushed back to the transmit BMA logic.
1	Flush Request Bit for Prefetch Buffer #1. 0 = Inactive 1 = Has data, needs to be flushed back to the transmit BMA logic.
0	Flush Service Request Bit for Prefetch Buffer #0. 0 = Inactive 1 = Has data, needs to be flushed back to the transmit BMA logic.

5.36 Receive BMA Packet Updated Info Register

32 bit R
0x1000 0124

After reset the contents of this register are indeterminate. After an Rx packet service request is issued, this register will hold values relevant to the current packet being pointed to in the "Receive BMA Buffer Service Information Register" on page 38.

Bit	Description
31:24	Updated Time-to-live
23:16	Reserved
15:0	Updated header checksum

5.37 Receive BMA Protocol 0 Identifier Register

32bit R/W
0x1000 012C

After reset, the register has an indeterminate value.

This register contains the value of a protocol identifier that is compared against bits [31:0] of the MAC header field in the Receive Packet Window.

Bit	Description
31:0	The value to check for, in the prefetched packet's MAC header.

5.38 Receive BMA Protocol 1 Identifier Register

32bit R/W
0x1000 0134

After reset, the register has an indeterminate value.

This register contains the value of a protocol identifier that is compared against bits [31:0] of the MAC header field in the Receive Packet Window.

Bit	Description
31:0	The value to check for, in the prefetched packet's MAC header.

5.39 Receive BMA Protocol 2 Identifier Register

32bit R/W
0x1000 013C

After reset, the register has an indeterminate value.

This register contains the value of a protocol identifier that is compared against bits [31:0] of the MAC header field in the Receive Packet Window.

Bit	Description
31:0	The value to check for, in the prefetched packet's MAC header.

5.40 Receive BMA Protocol 3 Identifier Register

32bit R/W
0x1000 0144

After reset, the register has an indeterminate value.

This register contains the value of a protocol identifier that is compared against bits [31:0] of the MAC header field in the Receive Packet Window.

Bit	Description
31:0	The value to check for, in the prefetched packet's MAC header.

5.41 FIB Root Register

32bit R/W
0x1000 014C

After reset the register has an indeterminate value.

This register is the Base Address for all MTRIE lookups.

Bit	Description
31:24	Reserved.
23:0	FIB Root address

5.42 Leaf Pointer Register

32bit R
0x1000 0154

After reset the value of this register is all zeros.

This register holds one of two values. For a successful MTRIE lookup with all checks passed, the value in this register is the leaf pointer for the current packet in service. The checks are:

1. THB = 0
2. Packet Length check passed
3. Packet is IPv4 with no-options
4. Identifiable protocol field in encapsulation MAC header
5. TTL is larger than 1
6. Checksum validation
7. MTRIE lookup was enabled at the time of lookup
8. No parity errors during lookup
9. A leaf was found within 3 lookup attempts.

If any of the checks fail, the value in this register is all zeros.

Bit	Description
31:1	bits [31:1] of the leaf pointer read from MTRIE lookup or all zeros
0	Always zero

5.43 Lookup Results Register

32bit R
0x1000 015C

After reset the value of this register is all zeros.

This is the last read value from MTRIE lookup, before the packet was given for service to the P4. If no errors occurred during hardware assist, this register will contain the Leaf pointer, equivalent to the "Leaf Pointer Register", above.

Bit	Description
31:0	bits [31:1] of the last read from MTRIE lookup

5.44 Port0 ACL Tree Base Register

32bit R/W
0x1000 0164

5.45 Port1 ACL Tree Base Register

32bit R/W
0x1000 016C

5.46 Port2 ACL Tree Base Register

32bit R/W
0x1000 0174

5.47 Port3 ACL Tree Base Register

32bit R/W
0x1000 017C

5.48 Port4 ACL Tree Base Register

32bit R/W
0x1000 0184

5.49 Port5 ACL Tree Base Register

32bit R/W
0x1000 018C

5.50 Port6 ACL Tree Base Register

32bit R/W
0x1000 0194

5.51 Port7 ACL Tree Base Register

32bit R/W
0x1000 019C

After reset the value of the above 8 registers will be zeros

Bit	Description
31:21	Contains the 13 bit based address of the ACL tree of that port
19:0	Reserved -- read as zeros

5.52 ACL Engine Config Register

16bit R/W
0x1000 01A4

After reset the value of this register is all zeros

Bit	Description
15:2	Reserved -- read as zeros
1	Port-info location in packet 0 - port information is located in bits [25:24] of word[5] of the BHDR 1 - port information is located in bits [8:6] of the 4byte POS-like MAC header
0	ACL Engine enable 0 - ACL engine is disabled 1 - ACL engine is enabled

5.53 Current ACL Node HI Bytes Register

32bit R
0x1000 01AC

After reset the value of this register is all zeros.

Bit	Description
31:0	Bits [63:32] of the most recently read ACL node

5.54 Current ACL Node LO Bytes Register

32bit R
0x1000 01B4

After reset the value of this register is all zeros.

Bit	Description
31:0	Bits [31:0] of the most recently read ACL node

5.55 Current ACL Hash Key Register

16bit R
0x1000 01BC

After reset the value of this register is all zeros.

Bit	Description
15:10	Reserved; Read as zeros
9:0	The value of the key used in the most recent ACL Hash table lookup

5.56 ACL Engine Status Register

32bit R
0x1000 01C4

After reset the value of this register is all zeros.

Bit	Description
31:4	Bits copied from the operand field of the STOP node
3	Permit/Deny bit. This bit is also copied off bit[3] of the operand field of the STOP node. 0 - Permit 1 - Deny
2	ACL Engine out-of-range address error 0 - No error 1 - An out-of-range DRAM address error has been encountered. See the "DRAM Catastrophic Error Address Register" for the offending address
1	ACL Engine uncorrected ECC error 0 - No error 1 - The ACL engine encountered a uncorrectable ECC error during a DRAM read. See the "DRAM Catastrophic Error Address Register" for address information
0	ACL Engine maxed out error 0 - The ACL engine completed successfully. 1 - The ACL engine did NOT reach a permit/deny node at the end of the maximum number of lookups

Bits [3:0] of this register are all zeros for a packet that has completed ACL lookup and is permitted.

5.57 ACL Hash Key LO Selection Register

32bit R/W
0x1000 01CC

After reset the value of this register is **32'h00000000**.

Bit	Description
31:24	Selection bits for bit[3] of the hash key.
22:16	Selection bits for bit[2] of the hash key.
15:8	Selection bits for bit[1] of the hash key.
7:0	Selection bits for bit[0] of the hash key.

These selection bits are used to construct the hashkey, one bit at a time. Each byte selects one of the 256 bits in the 32bytes immediately following the BHDR.

5.58 ACL Hash Key MID Selection Register

16bit R/W
0x1000 01D4

After reset the value of this register is **32'h0000**.

Bit	Description
31:24	Selection bits for bit[7] of the hash key.
22:16	Selection bits for bit[6] of the hash key.
15:8	Selection bits for bit[5] of the hash key.
7:0	Selection bits for bit[4] of the hash key.

These selection bits are used to construct the hashkey, one bit at a time. Each byte selects one of the 256 bits in the 32bytes immediately following the BHDR.

5.59 ACL Hash Key HI Selection Register

16bit R/W
0x1000 01DC

After reset the value of this register is **16'h0000**.

Bit	Description
15:8	Selection bits for bit[9] of the hash key.
7:0	Selection bits for bit[8] of the hash key.

These selection bits are used to construct the hashkey, one bit at a time. Each byte selects one of the 256 bits in the 32bytes immediately following the BHDR.

5.60 ACL Compare Mask 0 Register

32bit R/W
0x1000 01E4

5.61 ACL Compare Mask 1 Register

32bit R/W
0x1000 01EC

5.62 ACL Compare Mask 2 Register

32bit R/W
0x1000 01F4

5.63 ACL Compare Mask 3 Register

32bit R/W
0x1000 01FC

5.64 ACL Compare Mask 4 Register

32bit R/W
0x1000 0204

5.65 ACL Compare Mask 5 Register

32bit R/W
0x1000 020C

5.66 ACL Compare Mask 6 Register

32bit R/W
0x1000 0214

5.67 ACL Compare Mask 7 Register

32bit R/W
0x1000 021C

5.68 ACL Compare Mask 8 Register

32bit R/W
0x1000 0224

5.69 ACL Compare Mask 9 Register

32bit R/W
0x1000 022C

After reset the value of this register is all zeros.

Bit	Description
31:0	32 bit mask set used during the comparison of ACL Nodes. Each bit value of 1 masks the compare of the corresponding bit in the ACL node OPERAND field.

5.70 ACL Max Nodes Register

16bit R/W
0x1000 0234

After reset the value of this register is 0.

Bit	Description
15:10	Reserved. Read as zeros
9:0	Maximum number of ACL node lookups, before the packet is handed over to software, for completion.

As a special note for TTM projects, the following calculations suggest a value to program to this register:

```
TTM linecard TX rate....................................650Kpps
Salsa engine max performance............................850Kpps
Max per-packet delay that can be tolerated by Salsa
engine, before it becomes the bottleneck.................5000ns
Each ACL lookup takes 160ns (includes DRAM precharge)
Max Number of ACL lookups within this time...............32
```

5.71 ACL Current Count Register

16bit R
0x1000 023C

After reset the value of this register is 0.

Bit	Description
15:10	Reserved. Read as zeros
9:0	The actual number of ACL node lookups, before the current packet was passed on to software

5.72 ECC Status Register

16bit R
0x1000 0244

- After reset the value of this register is 0.

Bit	Description
15:2	Reserved. Read as zeros
1	Single-bit error on read 0 - No error 1 - Error occurred
0	Multiple-bit error on read 0 - No error 1 - Error occurred

All error status bits in this register clear upon read.

5.73 ECC Diagnostic Syndrome Register

8bit R/W
0x1000 024C

After reset the value of this register is 0.

Bit	Description
7:0	Provides direct write access to syndrome register. The contents of this register will be written to DRAM if bit [1] of the "ECC Config Register" has been set.

5.74 ECC Config Register

8bit R/W
0x1000 0254

After reset the value of this register is 8'b0000_0101.

Bit	Description
7:3	Reserved. Read as zeros

Bit	Description
2	Single bit error correction 0 - Disabled 1 - Enabled
1	Single bit error notification 0 - Single bit read errors will be silently corrected and forwarded to the processor 1 - Single bit read errors will cause an interrupt. Errors will be corrected as data is forwarded to the processor
0	Use hardware-generated syndrome 0 - The value in the "ECC Diagnostic Syndrome Register" is used during DRAM writes. This is NOT the normal mode of operation, and should only be used during diagnostic testing. 1 - A hardware-generated syndrome is used for writes to DRAM.

5.75 ECC SBE Address Register

32bit R
0x1000 025C

After reset the value of this register is all zeros.

Bit	Description
31:0	Contains the address of the first SBE since the "ECC Status Register" was last read.

5.76 ECC SBE Syndrome Register

8bit R
0x1000 0264

After reset the value of this register is all zeros.

Bit	Description
7:0	Contains the syndrome of the first SBE since the "ECC Status Register" was last read.

5.77 ECC MBE Syndrome Register

8bit R
0x1000 026C

After reset the value of this register is all zeros.

Bit	Description
7:0	Contains the syndrome of the first MBE since the "ECC Status Register" was last read.

Appendix A Perl program used to choose a hashing key.

To better understand this problem, a Perl program was written to calculate the number of checks a packet would encounter based on the hash key generated from its fields. The input to the perl program is a customer ACL. The results after this program are based on a smaller hash size of 256buckets, and was based on a 315 entry ACL from AOL.

```
#!/sw/current/hppabin/perl5

##################################################################
#                                                                #
# acl2tree.p   A perl5 program to traverse an ACL and determine #
#              what the worst delays would be for a hardware     #
#              tree lookup algorithm.                            #
#              acl2tree.p <ACL file>                             #
#                                                                #
##################################################################


# get the options handling package.
require "getopts.pl";
# define the options that we expect.
&Getopts('v');

# check that the options that we need have been set.
if ($#ARGV < 0) {
    print "
 ACL2TREE.p  A perl5 program to traverse an ACL and determine what the
             worst delays would be for a hardware tree lookup algorithm.
             acl2tree.p -v <ACL file>, where
                -v Produce verbose output
    \n";
    exit 1;
    }


use POSIX;

%hoprot = (
  udp => 17,
  tcp => 6,
  ip => 4,
  ipinip => 4,
  icmp => 1,
  igrp => 9,
  gre => 47,
  igmp => 2,
  nos => 94,
  ospf => 89,
  );

%holabels = (
```

```
    bgp => 179,
    biff => 512,
    bootpc => 68,
    bootps => 67,
    chargen => 19,
    cmd => 514,
    daytime => 13,
    discard => 9,
    dnsix => 195,
    domain => 53,
    echo => 7,
    "echo-reply" => 8,
    exec => 512,
    finger => 79,
    ftp => 21,
    "ftp-data" => 20,
    gopher => 70,
    hostname => 101,
    ident => 113,
    irc => 194,
    klogin => 543,
    kshell => 544,
    log => 513,
    login => 513,
    lpd => 515,
    mobile-ip => 434,
    nameserver => 42,
    "netbios-dgm" => 138,
    "netbios-ns" => 137,
    ntp => 118,
    nntp => 119,
    "packet-too-big" => 32,
    pop2 => 109,
    pop3 => 110,
    rip => 520,
    smtp => 25,
    snmp => 26,
    snmptrap => 27,
    sunrpc => 111,
    syslog => 514,
    tacacs => 49,
    talk => 517,
    telnet => 23,
    tftp => 221,
    time => 37,
    uucp => 540,
    whois => 43,
    www => 80,
    xdmcp => 177,
    );

####################################################################
#  Initializations                                                 #
####################################################################

$hwchecks = 2;
$filename = $ARGV[0];
$nbits = 88;
$bmax = 88;
```

```
$bmin = 88;
$bind = 16;
$outbits = 12;
$endhash = (2**$outbits);
for ($i=0; $i < $nbits; $i++) {
   $min[$i] = $i;
   $max[$i] = 0;
   $zerocount[$i] = 0;
   $onecount[$i] = 0;
   $outstr[$i] = '0';
}


######################################################################
#  Count the 0 and 1 occurrences of each of the 88 bits              #
#  by parsing file once                                              #
######################################################################

open (ACLFILE, $ARGV[0]) or die;

while (<ACLFILE>) {
   if ($_ !~ /^[ ]*access-list/) {
      #silently ignore line
      next;
      }
   elsif (!(/udp/ || /tcp/ || /ip/ || /icmp/ || /igrp/ || /gre/ ||
        /igmp/ || /nos/ || /ospf/)) {
      print ("ERROR: Unknown protocol type in next line!\n$_\n");
      die;
      }
   #implicit else condition is to continue..
   chop;
   @aclarr = parse_current_line($_);


   # Hari's code...
   # cycle from 0 to nbits and update zero/one counts based on (0,1,x)
   # also keep track of min/max of zero/one counts
   for ($i = 0; $i < $nbits; $i++) {
       if ($aclarr[$i] eq 'x'){
          $zerocount[$i]++;
          $onecount[$i]++;
       } elsif ($aclarr[$i] eq '0'){
          $zerocount[$i]++;
       } elsif ($aclarr[$i] eq '1'){
          $onecount[$i]++;
       } else {
          print ("hash_min_max: Unknown character aclarr[$i] = $aclarr[$i]\n");
          die;
       }
       $min[$i] = ($zerocount[$i] < $onecount[$i]) ?
                   $zerocount[$i] : $onecount[$i];
       $max[$i] = ($zerocount[$i] > $onecount[$i]) ?
                   $zerocount[$i] : $onecount[$i];
       }
   }


######################################################################
```

```
# Determine which bits occurred as 0 and 1 most evenly. These are    #
# good candidates for the key. (Hari's code)                         #
######################################################################

# Select best bits with min/max values
for ($i = 0; $i < $nbits; $i++) {
#
# concatenate bitstrings(16 bits) of max, min, index
# sort them and then use index to set the outstr bits correctly
#
   $fixmax = substr((unpack("B*", pack("n", $max[$i]))), -$bind, $bind);
   $fixmin = substr((unpack("B*", pack("n", $min[$i]))), -$bind, $bind);
   $fixind = substr((unpack("B*", pack("n", $i))), -$bind, $bind);
   $presort[$i] = $fixmax.$fixmin.$fixind;
   print("for max=$max[$i], min=$min[$i] and i=$i: $presort[$i]\n");
   print("   where, fixmax = $fixmax, fixmin=$fixmin\n");
}
@postsort = sort(@presort);
#
# get index from post sorted array and set those bits to 1 in outstr
#
for ($i=0; $i < $outbits; $i++) {
   $bitpos = bintodec(substr($postsort[$i], -$bind, $bind));
   $outstr[$bitpos] = '1';
}

# print min/max results
#
foreach $w (@outstr) {
   $bitmask = $bitmask.$w;
}
print ("Selected bitmask=$bitmask\n");

#
# Extra prints that can be put under verbose
#
if ($opt_v) {
   print ("\nMin=");
   for ($i=0; $i < $nbits; $i++) {
      print("$i=$min[$i] ");
      }
   print ("\nMax=");
   for ($i=0; $i < $nbits; $i++) {
      print("$i=$max[$i] ");
      }
   print ("\nOnecount=");
   for ($i=0; $i < $nbits; $i++) {
      print("$i=$onecount[$i] ");
      }
   print ("\nZerocount=");
   for ($i=0; $i < $nbits; $i++) {
      print("$i=$zerocount[$i] ");
      }
   print ("\n");
   }


######################################################################
# Using the 8 selected bits, form a key for each ACL entry that      #
```

```
# defines where in the hash table to add that entry                #
####################################################################

close (ACLFILE);
open (ACLFILE, $ARGV[0]) or die;

while (<ACLFILE>) {
   if ($_ !~ /^[ ]*access-list/) {
      #silently ignore line
      next;
      }
   #implicit else condition is to continue..
   chop;
   @aclarr = &parse_current_line($_);

   $dongle = "";
   for ($i=0; $i<$nbits; $i++) {
      if ($outstr[$i]) {
         $dongle = $aclarr[$i].$dongle;
         }
      }

####################################################################
# walk thru the hash table, incrementing bins that fits the key    #
####################################################################

#cycle from 0 theu 255, and see what fits the dongle mask

for ($i = 0; $i < $endhash; $i++) {
    $bini = substr((unpack("B*", pack("n", $i))), -$outbits, $outbits);
    $fits_into_mask = 1;
    @aobini = split(//,$bini);
    @aodongle = split(//,$dongle);
    for ($j = 0; $j <$outbits; $j++) {
       if (($aobini[$j] ne $aodongle[$j]) && ($aodongle[$j] ne 'x')) {
          $fits_into_mask = 0;
          }
       }
    if ($fits_into_mask == 1) {
       $aoacls[$i] = $aoacls[$i] + $hwchecks;
       }
    }
}


#################################################################
#     Print out the hash table, and each bin's total checks     #
#################################################################

$max = 0; $min = 99999; $total = 0; $rows = ($endhash/8);
for ($i = 0; $i < ($rows-1); $i++) {
   $~ = 'REPORT2';
   if ($opt_v) {
      write;
      }
   for ($j = 0; $j < 8; $j++) {
      $max = ($aoacls[$i*8+$j] > $max) ? $aoacls[$i*8+$j] : $max;
      $min = ($aoacls[$i*8+$j] < $min) ? $aoacls[$i*8+$j] : $min;
      $total = $total + $aoacls[$i*8+$j];
```

```
      }
   }
$average = $total/$endhash;
print ("ACL2TREE ran on $filename\n");
print "Maximum checks: $max\nMinimum checks: $min\nAverage checks:$average\n";


###################################################################
# Formats for various screen dumps                                #
###################################################################

format STDOUT_TOP =
  SourceAddr       DestAddr         Proto.   Port         StartPort  Key
  --------------   --------------   -------  -------      ---------  --------
.

format STDOUT =
  @<<<<<<<<<<<<<   @<<<<<<<<<<<<<   @<<<<<<  @<<<<<<      @<<<<<<<<  @<<<<<<<<<<
  $fullsa,         $fullda,         $protocol,$porttype,  $startport,$dongle
.

format REPORT2 =
  @>>>:@<<< @>>>:@<<< @>>>:@<<< @>>>:@<<< @>>>:@<<< @>>>:@<<< @>>>:@<<< @>>>:@<<<
  ($i*8+0), $aoacls[$i*8+0],($i*8+1), $aoacls[$i*8+1],($i*8+2),
$aoacls[$i*8+2],($i*8+3), $aoacls[$i*8+3],($i*8+4), $aoacls[$i*8+4],($i*8+5),
$aoacls[$i*8+5],($i*8+6), $aoacls[$i*8+6],($i*8+7), $aoacls[$i*8+7]
.

###################################################################
# Function definitions                                            #
###################################################################


# function to calculate start port

sub calc_start_port {
   local($port) = @_;
   if (!$port) {
      return "any";
      }
   elsif (isdigit($port)) {
      return $port;
      }
   elsif ($holabels{$port}) {
      return $holabels{$port};
          }
   else {
      print "Error! Can't understand a port name of $port\n";
      die;
      }
   }


# A function to parse the current ACL entry, and return an array
# of values for SA, SA-MASK, DA, DA-MASK, PROTOCOL, DEST. PORT

sub parse_current_line{
   local($line) = @_;
   my @aclarr;
```

```
@word = split(" ",$line);
#getting protocol type
while (($w = shift(@word))
       && ($w !~ /udp/)
       && ($w !~ /tcp/)
       && ($w !~ /ip/)
       && ($w !~ /icmp/)
       && ($w !~ /igrp/)
       && ($w !~ /gre/)
       && ($w !~ /igmp/)
       && ($w !~ /nos/)
       && ($w !~ /ospf/)
       ) {
   };
$protocol = $hoprot{$w};

#getting source address
$w = shift(@word);
if ($w =~ /host/) {       #access-list 141 permit ip host 152.163.177.207
   $fullsa = shift(@word);
   $fullsamask = "0.0.0.0";
   }
elsif ($w =~ /any/) {     #access-list 141 permit tcp any
   $fullsa = $w;
   }
else {                    #access-list 141 permit tcp 198.81.7.0 0.0.0.255
   $fullsa = $w;
   $fullsamask = shift(@word);
   }
#getting destination address
$w = shift(@word);
if ($w =~ /host/) {       #access-list 141 permit ip host 152.163.177.207
   $fullda = shift(@word);
   $fulldamask = "0.0.0.0";
   }
elsif ($w =~ /any/) {     #access-list 141 permit tcp any
   $fullda = $w;
   }
else {                    #access-list 141 permit tcp 198.81.7.0 0.0.0.255
   $fullda = $w;
   $fulldamask = shift(@word);
   }
#getting port info
$porttype = "";
$startport = "any";
while ($w = shift(@word)){
   if ($w =~ /range/){
      $porttype = $w;
      $w = shift(@word);
      $startport = &calc_start_port($w);
      shift(@word);
      }
   elsif (($w =~ /gt/) | ($w =~ /eq/) | ($w =~ /lt/)) {
      $porttype = $w;
      $w = shift(@word);
      $startport = &calc_start_port($w);
      }
   elsif ($w =~ /established/){
      $porttype = $w;
```

```
        $startport = 1234;
        }
     else {
        $porttype = 'eq';
        $startport = &calc_start_port($w);
        }
     }

  ($sa[0],$sa[1],$sa[2],$sa[3]) = split(/\./, $fullsa);
  ($samask[0],$samask[1],$samask[2],$samask[3]) = split(/\./, $fullsamask);
  ($da[0],$da[1],$da[2],$da[3]) = split(/\./, $fullda);
  ($damask[0],$damask[1],$damask[2],$damask[3]) = split(/\./, $fulldamask);

if ($opt_v) {
   #not the best place to put this write, since $dongle hasn't been
   #define for the current ACL entry.
   write;
   }

#Convert these fields to binary, pushing them LSB first, into an array.

#Convert SA to binary. Use x's for any or for any bit that's masked
for ($j = 3; $j >= 0; $j--){
  @sabits = split(//,substr((unpack("B*", pack("n", $sa[$j]))), -8, 8));
  @smbits = split(//,substr((unpack("B*", pack("n", $samask[$j]))), -8, 8));
  for ($i=7; $i>=0; $i--) {
    $bit = ($smbits[$i] || ($fullsa =~ /any/))? 'x' : $sabits[$i];
    push(@aclarr,$bit);
    }
  }

#Convert dA to binary. Use x's for any or for any bit that's masked
for ($j = 3; $j >= 0; $j--){
  @dabits = split(//,substr((unpack("B*", pack("n", $da[$j]))), -8, 8));
  @dmbits = split(//,substr((unpack("B*", pack("n", $damask[$j]))), -8, 8));
  for ($i=7; $i>=0; $i--) {
    $bit = ($dmbits[$i] || ($fullda =~ /any/))? 'x' : $dabits[$i];
    push(@aclarr,$bit);
    }
  }

#Convert protocol to binary.
@protbits = split(//,substr((unpack("B*", pack("n", $protocol))), -8, 8));
for ($i=7; $i>=0; $i--) {
   $bit = $protbits[$i];
   push(@aclarr,$bit);
   }

#Convert port to binary.
@portbits = split(//,substr((unpack("B*", pack("n", $startport))), -16, 16));
for ($i=15; $i>=8; $i--) {
   $bit = ($porttype =~ /eq/) ? $portbits[$i] : 'x';
   push(@aclarr,$bit);
   }
for ($i=7; $i>=0; $i--) {
   $bit = $portbits[$i];
   push(@aclarr,$bit);
   }
```

```
   return @aclarr;
   }


# A function to convert from binary to decimal

sub bintodec {
   unpack("N", pack("B32", substr("0" x 32 . shift, -32)));
}
```

Experiment #1. Hashing on the 2nd octet of the destination address:

0:56	1:56	2:56	3:56	4:56	5:56	6:56	7:56
8:56	9:56	10:56	11:57	12:56	13:56	14:56	15:56
16:116	17:185	18:56	19:56	20:56	21:56	22:56	23:56
24:56	25:56	26:56	27:56	28:56	29:56	30:56	31:56
32:56	33:56	34:56	35:56	36:56	37:56	38:56	39:56
40:56	41:56	42:56	43:56	44:56	45:56	46:56	47:56
48:61	49:57	50:57	51:65	52:58	53:66	54:57	55:57
56:57	57:57	58:57	59:80	60:57	61:57	62:57	63:57
64:56	65:56	66:56	67:56	68:56	69:57	70:63	71:56
72:56	73:56	74:56	75:56	76:56	77:56	78:56	79:56
80:56	81:56	82:56	83:56	84:56	85:56	86:56	87:56
88:56	89:56	90:56	91:56	92:56	93:56	94:56	95:56
96:56	97:56	98:56	99:56	100:56	101:56	102:56	103:57
104:56	105:56	106:56	107:56	108:56	109:56	110:56	111:56
112:56	113:56	114:56	115:56	116:56	117:56	118:56	119:56
120:56	121:56	122:56	123:56	124:56	125:56	126:56	127:56
128:56	129:56	130:56	131:56	132:56	133:56	134:58	135:56
136:56	137:56	138:56	139:56	140:56	141:56	142:56	143:56
144:56	145:56	146:56	147:56	148:56	149:56	150:56	151:56
152:57	153:56	154:56	155:56	156:56	157:56	158:56	159:56
160:56	161:56	162:56	163:56	164:56	165:56	166:56	167:56
168:56	169:56	170:56	171:56	172:56	173:56	174:56	175:56
176:56	177:56	178:56	179:56	180:56	181:56	182:56	183:56
184:56	185:56	186:56	187:59	188:56	189:56	190:56	191:56
192:56	193:56	194:56	195:56	196:56	197:56	198:56	199:56
200:56	201:56	202:56	203:56	204:56	205:56	206:56	207:56
208:56	209:56	210:56	211:56	212:56	213:56	214:56	215:56
216:56	217:56	218:56	219:56	220:56	221:56	222:56	223:56
224:56	225:56	226:56	227:56	228:56	229:56	230:56	231:56
232:56	233:56	234:56	235:56	236:56	237:56	238:56	239:56
240:56	241:56	242:56	243:56	244:56	245:56	246:56	247:56
248:56	249:56	250:56	251:56	252:56	253:56	254:56	255:56

Maximum checks:185, Minimum checks:56

Experiment #2. Hashing on an 8bit combination of source address (4 bits of octet 2) and destination address (4 bits of octet 2):

0:12	1:119	2:11	3:30	4:16	5:11	6:11	7:11
8:11	9:11	10:11	11:14	12:11	13:11	14:11	15:11
16:29	17:138	18:29	19:65	20:34	21:29	22:29	23:29

24:29	25:29	26:29	27:32	28:29	29:29	30:29	31:29
32:11	33:119	34:11	35:30	36:16	37:11	38:11	39:11
40:11	41:11	42:11	43:14	44:11	45:11	46:11	47:11
48:34	49:221	50:34	51:61	52:40	53:34	54:34	55:34
56:36	57:34	58:34	59:37	60:34	61:34	62:34	63:34
64:11	65:119	66:11	67:30	68:17	69:11	70:11	71:11
72:11	73:11	74:11	75:14	76:11	77:11	78:11	79:11
80:11	81:119	82:11	83:30	84:16	85:11	86:11	87:11
88:11	89:11	90:11	91:14	92:11	93:11	94:11	95:11
96:11	97:119	98:11	99:30	100:17	101:11	102:12	103:11
104:11	105:11	106:11	107:14	108:11	109:11	110:11	111:11
112:11	113:119	114:11	115:30	116:16	117:11	118:11	119:11
120:11	121:11	122:11	123:14	124:11	125:11	126:11	127:11
128:12	129:120	130:12	131:33	132:17	133:12	134:12	135:12
136:12	137:12	138:12	139:15	140:12	141:12	142:12	143:12
144:11	145:119	146:11	147:30	148:16	149:11	150:11	151:11
152:11	153:12	154:11	155:14	156:11	157:11	158:11	159:11
160:11	161:119	162:11	163:30	164:16	165:11	166:11	167:11
168:11	169:11	170:11	171:14	172:11	173:11	174:11	175:11
176:14	177:122	178:14	179:33	180:19	181:14	182:14	183:14
184:14	185:14	186:14	187:17	188:14	189:14	190:14	191:14
192:11	193:119	194:11	195:30	196:16	197:11	198:11	199:11
200:11	201:11	202:11	203:14	204:11	205:11	206:11	207:11
208:11	209:119	210:11	211:30	212:16	213:11	214:11	215:11
216:11	217:11	218:11	219:14	220:11	221:11	222:11	223:11
224:11	225:119	226:11	227:30	228:16	229:11	230:11	231:11
232:11	233:11	234:11	235:14	236:11	237:11	238:11	239:11
240:11	241:119	242:11	243:30	244:16	245:11	246:11	247:11
248:11	249:11	250:11	251:14	252:11	253:11	254:11	255:11

Maximum checks:221, Minimum checks:11

<u>Experiment #3: Hashing on an 8 bit combination of destination address (4 bits of octet 2) and destination port (lower 4 bits):</u>

0:4	1:54	2:4	3:23	4:7	5:4	6:4	7:4
8:5	9:4	10:4	11:4	12:4	13:4	14:4	15:4
16:4	17:61	18:4	19:9	20:6	21:4	22:4	23:4
24:5	25:4	26:4	27:4	28:4	29:4	30:4	31:4
32:6	33:56	34:6	35:11	36:8	37:6	38:6	39:6
40:7	41:6	42:6	43:6	44:6	45:6	46:6	47:6
48:3	49:60	50:2	51:7	52:4	53:2	54:3	55:2
56:3	57:3	58:2	59:2	60:2	61:2	62:2	63:2
64:2	65:51	66:2	67:7	68:4	69:2	70:2	71:2
72:3	73:2	74:2	75:2	76:2	77:2	78:2	79:2
80:22	81:67	82:22	83:27	84:25	85:22	86:22	87:22
88:23	89:22	90:22	91:24	92:22	93:22	94:22	95:22
96:2	97:59	98:2	99:11	100:4	101:2	102:2	103:2
104:4	105:2	106:2	107:2	108:2	109:2	110:2	111:2
112:7	113:64	114:7	115:12	116:10	117:7	118:7	119:7
120:8	121:7	122:7	123:7	124:7	125:7	126:7	127:7
128:2	129:51	130:2	131:7	132:4	133:2	134:2	135:2
136:3	137:2	138:2	139:2	140:2	141:2	142:2	143:2
144:3	145:46	146:3	147:8	148:6	149:3	150:3	151:3
152:4	153:3	154:3	155:3	156:3	157:3	158:3	159:3
160:16	161:59	162:16	163:22	164:19	165:16	166:16	167:16
168:17	169:16	170:16	171:16	172:16	173:16	174:16	175:16
176:2	177:51	178:2	179:10	180:4	181:2	182:2	183:2

184:3	185:2	186:2	187:2	188:2	189:2	190:2	191:2
192:2	193:44	194:2	195:7	196:4	197:2	198:2	199:2
200:3	201:2	202:2	203:2	204:2	205:2	206:2	207:2
208:2	209:51	210:2	211:11	212:4	213:2	214:2	215:2
216:3	217:2	218:2	219:2	220:2	221:2	222:2	223:2
224:2	225:58	226:2	227:7	228:4	229:2	230:2	231:2
232:3	233:2	234:2	235:2	236:2	237:2	238:2	239:2
240:8	241:72	242:8	243:28	244:11	245:8	246:8	247:8
248:9	249:8	250:8	251:9	252:8	253:8	254:8	255:8

Maximum checks:72, Minimum checks:2

Appendix B Prefetch Packet Header Format

```
31  29  27  25  23  21  19  17  15  13  11  09  07  05  03  01
  30  28  26  24  22  20  18  16  14  12  10  08  06  04  02  00

BHDR: 8 x 4Bytes
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                              NA                               | 0
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                              NA                               | 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                              NA                               | 2
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                              NA                               | 3
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                              NA                               | 4
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|T| Input Info. |       NA      |            length             | 5
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                              NA                               | 6
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                              NA                               | 7
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

MAC header: 1 x 4Bytes
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|    Address    |    Control    |           Protocol            | 8
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

IP header: 5 x 4Bytes
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Version|  IHL  |Type of Service|          Total Length         | 9
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|         Identification        |Flags|      Fragment Offset    | 10
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Time to Live |    Protocol   |         Header Checksum       | 11
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Source Address                          | 12
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Destination Address                        | 13
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Payload: 2 x 4Bytes
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        | 14
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                              Data                             | 15
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Please note the following allowed values for each of the 5 bit decodes:

Table 5: A Summary of the selection choices for the ACL hash key

Selection Bits	KEY [7:6]	KEY [5:4]	KEY [3:2]	KEY [1:0]
00	SA[7:6]	SA[5:4]	SA[3:2]	SA[1:0]
01	SA[15:14]	SA[13:12]	SA[11:10]	SA[9:8]
02	SA[23:22]	SA[21:20]	SA[19:18]	SA[17:16]
03	SA[31:30]	SA[29:28]	SA[27:26]	SA[25:24]
04	DA[7:6]	DA[5:4]	DA[3:2]	DA[1:0]
05	DA[15:14]	DA[13:12]	DA[11:10]	DA[9:8]
06	DA[23:22]	DA[21:20]	DA[19:18]	DA[17:16]
07	DA[31:30]	DA[29:28]	DA[27:26]	DA[25:24]
08	SP[7:6]	SP[5:4]	SP[3:2]	SP[1:0]
09	SP[15:14]	SP[13:12]	SP[11:10]	SP[9:8]
0a	DP[7:6]	DP[5:4]	DP[3:2]	DP[1:0]
0b	DP[15:14]	DP[13:12]	DP[11:10]	DP[9:8]
0c	PROT[7:6]	PROT[5:4]	PROT[3:2]	PROT[1:0]
0d	TOS[7:6]	TOS[5:4]	TOS[3:2]	TOS[1:0]
0e	SA[3:2]	SA[1:0]	SA[7:6]	SA[5:4]
0f	SA[11:10]	SA[9:8]	SA[15:14]	SA[13:12]
10	SA[19:18]	SA[17:16]	SA[23:22]	SA[21:20]
11	SA[27:26]	SA[25:24]	SA[31:30]	SA[29:28]
12	DA[3:2]	DA[1:0]	DA[7:6]	DA[5:4]
13	DA[11:10]	DA[9:8]	DA[15:14]	DA[13:12]
14	DA[19:18]	DA[17:16]	DA[23:22]	DA[21:20]
15	DA[27:26]	DA[25:24]	DA[31:30]	DA[29:28]
16	SP[3:2]	SP[1:0]	SP[7:6]	SP[5:4]
17	SP[11:10]	SP[9:8]	SP[15:14]	SP[13:12]
18	DP[3:2]	DP[1:0]	DP[7:6]	DP[5:4]
19	DP[11:10]	DP[9:8]	DP[15:14]	DP[13:12]
1a	PROT[3:2]	PROT[1:0]	PROT[7:6]	PROT[5:4]
1b	TOS[3:2]	TOS[1:0]	TOS[7:6]	TOS[5:4]
1c	Reserved	Reserved	Reserved	Reserved
1d	Reserved	Reserved	Reserved	Reserved
1e	Reserved	Reserved	Reserved	Reserved
1f	Reserved	Reserved	Reserved	Reserved

Suggested values for this register are:

18_18_6_6 Key = {DP[3:2],DP[1:0],DA[19:18],DA[17:16]}

0e_18_6_6 Key = {SA[3:2],DP[1:0],DA[19:18],DA[17:16]}